Static Analysis and Dynamic Adaptation of Parallelism.

Pierre Huchant

université de BORDEAUX

Thèse présentée
pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

École doctorale de Mathématique et Informatique de Bordeaux
Spécialité : **Informatique**

Par **Pierre HUCHANT**

Analyse statique et adaptation dynamique du parallélisme

Après avis de :

Cédric Bastoul	Professeur, Université de Strasbourg	Rapporteur
William Jalby	Professeur, UVSQ	Rapporteur

Soutenue le 29 mars 2019 devant la commission d'examen composée de :

Martin Quinson	Professeur, ENS Rennes	Président du jury
Denis Barthou	Professeur, Bordeaux INP	Directeur de thèse
Marie-Christine Counilh	Maître de conférence, Université de Bordeaux	Examinatrice
Cédric Bastoul	Professeur, Université de Strasbourg	Rapporteur
William Jalby	Professeur, UVSQ	Rapporteur

Résumé

Les applications de calcul scientifique ont besoin de ressources de calcul de plus en plus importantes et beaucoup de grands challenges scientifiques exigent des capacités de calcul Exascale (10 puissance 18 calculs par seconde) pour être relevés. L'un des principaux obstacles pour atteindre l'Exascale est la difficulté de programmer les architectures parallèles actuelles. De nouvelles méthodes automatiques sont nécessaires pour combler l'écart entre les développeurs d'applications scientifiques et les experts en calcul haute performance. De plus, les applications scientifiques devenant de plus en plus complexes et étant supposées s'exécuter à très large échelle, de nouveaux outils sont nécessaires pour aider les développeurs lors de la phase de débogage des programmes. Cette thèse explore la combinaison de méthodes statiques et dynamiques pour faciliter la programmation des applications de calcul haute performance. Deux enjeux majeurs sont étudiés : faciliter la programmation des architectures hétérogènes et prévenir les interblocages dans les programmes parallèles.

La première partie de cette thèse s'intéresse à l'adaptation automatique des tâches de calcul aux architectures hétérogènes. Nous proposons une nouvelle méthode pour faciliter la programmation des architectures hétérogènes composées de plusieurs unités de calcul (CPUs et GPUs). Le programmeur exprime le parallélisme de son application sous forme de tâches OpenCL sans se soucier des problèmatiques liées à l'architecture sur laquelle son code sera exécuté. Ensuite notre méthode partitionne automatiquement chaque tâche en sous-tâches et équilibre la charge de travail entre les unités de calcul afin de tirer pleinement avantage de toutes les ressources de calcul de la machine.

La deuxième partie de cette thèse porte sur la détection automatique des interblocages dans les programmes parallèles. Nous proposons une nouvelle analyse statique permettant de détecter précisement les chemins d'exécution menant à des interblocages dans les programmes parallèles. Cette analyse statique est ensuite combinée à une instrumentation dynamique du code afin de prévenir les interblocages à l'exécution.

Les solutions proposées dans cette thèse ont été testée et validées sur des cas réels d'applications parallèles.

Mots clés: calcul haute performance, architectures hétérogènes, débogage, analyse statique, analyse dynamique, OpenCL, équilibrage de charge

Abstract

Scientific applications have an increasing need of resources and many grand scientific challenges require exascale compute capabilities to be addressed. One major concern to achieve exascale is programmability. New automatic methods are required to fill the gap between developers of scientific applications and HPC experts. In addition, as scientific applications are becoming more and more complex and are supposed to run at extreme scale, new tools are required to assist developers in the debugging phase of application development. This thesis explores the combination of static and dynamic methods to improve programmability of HPC applications. Two major issues are investigated: the complexity of programming heterogeneous architectures and the prevention of deadlocks in parallel programs.

The first part of this thesis investigates the automatic task adaptation for heterogeneous architectures. More precisely, we propose a new method to improve programmability of heterogeneous architectures. The programmer expresses the parallelism of his application through a sequence of OpenCL tasks without considering issues related to the underlying architecture where its code will be executed. Then our method automatically partitions the tasks into sub-tasks executed by each device and handles load balancing between the devices to take full advantage of the machine capabilities.

The second part of this thesis investigates the automatic detection and prevention of deadlocks in parallel programs. We propose a novel static analysis to precisely detect execution paths in parallel programs potentially leading to deadlocks. This static analysis is then combined with a dynamic instrumentation of the code to automatically prevent deadlocks at runtime.

The solutions proposed in this thesis have been tested and validated on real parallel applications.

Key words: High Performance Computing, Heterogeneous Architectures, Debugging, Static Analysis, Dynamic Analysis, OpenCL, Load Balancing

Contents

1 Introduction **9**
- 1.1 Introduction to High Performance Computing . . . 9
  - 1.1.1 Parallel Architectures Evolution . . . 9
  - 1.1.2 Towards Exascale . . . 10
  - 1.1.3 Challenges for Programmability . . . 11
- 1.2 Expressing Parallelism . . . 12
  - 1.2.1 OpenMP . . . 13
  - 1.2.2 MPI . . . 13
  - 1.2.3 Partitioned Global Address Space . . . 14
  - 1.2.4 Programming Many-Core . . . 14
  - 1.2.5 Summary . . . 16
- 1.3 Debugging HPC Applications . . . 16
  - 1.3.1 Definitions . . . 16
  - 1.3.2 Parallel Debugging . . . 17
  - 1.3.3 Bug Detection Techniques . . . 18
  - 1.3.4 Summary . . . 20
- 1.4 Outline and Contributions . . . 20

I Automatic Task Adaptation for Heterogeneous Architectures **23**

2 Automatic Adaptation for Iterated Sequences of Irregular Kernels **25**
- 2.1 Context and Parallelization Model . . . 26
  - 2.1.1 Context . . . 26
  - 2.1.2 Parallelization Model . . . 29
  - 2.1.3 Partitioning Strategies . . . 31
- 2.2 Challenges . . . 33
  - 2.2.1 Data Partitioning . . . 34
  - 2.2.2 Load Balancing . . . 38
  - 2.2.3 Summary . . . 42
- 2.3 Principle of Adaptive Partitioning . . . 42
  - 2.3.1 Static Analysis and Transformation . . . 44
  - 2.3.2 Dynamic Adaptation . . . 45
  - 2.3.3 Buffer Management . . . 45
  - 2.3.4 General Algorithm . . . 46
- 2.4 Summary . . . 47

3 Automatic Data Partitioning **49**
3.1 Memory Region Analysis . . . 50
3.1.1 Objectives and Principles . . . 50
3.1.2 Parametric Region Construction and Instantiation . . . 52
3.1.3 Overlapping Write Regions . . . 55
3.1.4 Atomics . . . 57
3.1.5 Limits . . . 57
3.2 Case Study: SOTL . . . 58
3.2.1 Algorithm and Data Structures . . . 59
3.2.2 Kernels Analysis . . . 62
3.2.3 Evaluation . . . 69
3.3 Related Work . . . 70
3.4 Summary . . . 72

4 Dynamic Load Balancing of Iterated Sequences of Irregular Kernels **75**
4.1 Load Balancing Computation . . . 76
4.1.1 Formalization . . . 76
4.1.2 Resolution Method . . . 77
4.1.3 Evaluation . . . 78
4.2 Communication-Aware Load Balancing . . . 81
4.2.1 Formalization . . . 81
4.2.2 Evaluation . . . 85
4.3 Related Work . . . 88
4.3.1 Static Coarse-grained Partitioning . . . 88
4.3.2 Dynamic Coarse-grain Partitioning . . . 90
4.3.3 Fine-grained Partitioning . . . 91
4.4 Implementation . . . 91
4.5 Summary and conclusion of the first part . . . 95

II Detection of Collective Errors Origin in Parallel Applications **97**

5 PARCOACH Extension for a Full-Interprocedural Collectives Verification **99**
5.1 Objectives and Principles . . . 100
5.1.1 PARCOACH . . . 100
5.1.2 Objective of our Full-Interprocedural Analysis . . . 107
5.2 Full-Interprocedural Analysis . . . 107
5.2.1 PPCFG Construction . . . 107
5.2.2 Collective Error Detection . . . 108
5.3 Code Instrumentation . . . 109
5.4 Experimental Results . . . 111
5.4.1 Static Analysis Results . . . 111
5.4.2 Execution Results . . . 114
5.5 Summary . . . 115

6 Multi-Valued Expression Analysis for Collective Checking **117**
6.1 Principle and Objective . . . 117
6.1.1 Challenges . . . 118
6.1.2 Principle . . . 119
6.2 Multi-Valued Expression Detection . . . 120
6.2.1 Enhanced SSA . . . 120
6.2.2 PDCG: Program Data- and Control-flow Dependence Graph . . . 121
6.2.3 Finding Collective Errors . . . 122
6.2.4 Example . . . 123
6.3 Related Works . . . 124
6.3.1 Dependence Analyses Techniques . . . 124
6.3.2 Collective Error Detection Techniques . . . 125
6.4 Experimental Results . . . 126
6.5 Summary and conclusion of the second part . . . 129

7 Conclusion and perspectives **131**
7.1 Automatic Tasks Adaptation for Heterogeneous Architectures . . . 131
7.1.1 Contributions Summary . . . 131
7.1.2 Perspectives . . . 132
7.2 Detection of Collective Errors Origins in Parallel Programs . . . 134
7.2.1 Contributions Summary . . . 134
7.2.2 Perspectives . . . 134

Bibliography **139**

List of Figures **151**

List of Tables **155**

Chapter 1

Introduction

The purpose of this chapter is to set the context of this thesis. **Section 1.1** introduces the main concepts of high performance computing. The section describes the evolution of parallel architectures, explores the impact involved by exascale and ends with the challenges and issues targeted in the scope of this thesis. **Section 1.2** presents the different programming models and languages to express parallelism. **Section 1.3** introduces the key concept and techniques for parallel debugging and presents state-of-the-art methods for bug detection in HPC applications. Finally, **Section 1.4** introduces the contributions of this thesis.

1.1 Introduction to High Performance Computing

Scientific applications like meteorology, nuclear physics or computational chemistry have an increasing need of resources to widen the domain of their simulations and increase the precision of their results. For many scientific fields, progress is impossible without the use of high performance computing [1]. To sustain this growing need, supercomputers have been developed. Nowadays, massively parallel architectures are omnipresent in high performance computing.

1.1.1 Parallel Architectures Evolution

For many years, thanks to the increase of the clock speed and the Moore's law which predicts that the number of transistors in processors double every two years, programmers did not have to modify their code to enhance the performances of their applications. However, due to heat dissipation and energy consumption issues, this period has come to an end around 2003. Since then, the trend has been towards the conception of microprocessors with multiple processing units known as cores. This allows for a significant increase in computational performance with more work in return for the programmer since the speedup on parallel architectures is limited by the sequential parts of the application (cf Amdahl's law).

Nowadays, there are two approaches. The first one, denoted as *multi-core* approach integrates a few cores (from two to several dozen) into a single microprocessor. Current laptops and desktops integrate this kind of processor. The second, *many-core* approach uses a large number of cores (several hundred). An example of this approach corresponds to the Graphical Processing Units (GPUs). The first attempts at using GPUs for non-graphics computation date back to 2005 and used corner cases of the graphics APIs [2]. Since then, GPUs have become very popular in scientific computing as accelerators thanks to a high memory bandwidth, a strong capacity to perform

floating-point operations and a low cost. In 2012 Intel also proposed its own accelerator, the MIC (Many Integrated Core) called Intel Xeon Phi.

Architectures composed of a multi-core processor and some many-core accelerators are called heterogeneous architectures. These architectures are omnipresent in today's supercomputers. Twice a year, the most powerful supercomputers in the world are tested and ordered based on their Linpack performance[1] in floating-point operations per second (flops). In the Top500 list of November 2018 [3], six of the top ten supercomputers contains accelerators and in the Green500 list [4] which order supercomputers based on their energy efficiency, all supercomputers from the top ten have many-core accelerators. Supercomputers consist in a thousand of nodes interconnected and each node usually contains multiple accelerators. For example, the top supercomputer in the Top500 list of November 2018 has 4,608 nodes and each node owns two IBM POWER9 multi-core processors and six NVIDIA Volta V100s GPUs.

Programming heterogeneous architectures is very hard as codes executed on accelerators (kernel codes) are often intrinsically difficult to write because of the complexity of the accelerator architecture. In addition, writing the code to manage the accelerators (host code executed by the host multi-core) is very cumbersome. Host codes have to explicitly manage memory allocations on each accelerator as well as transfers between host memory and accelerator memories. Moreover, the load must be balanced between GPUs and CPUs in order to leverage the computing power of heterogeneous architectures.

1.1.2 Towards Exascale

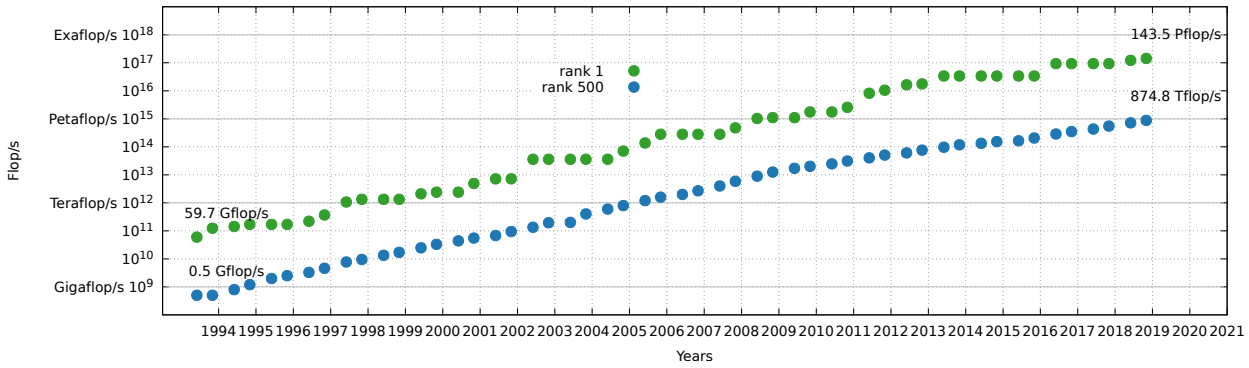
Over the last 30 years, the computing power of supercomputers has grown exponentially. **Figure 1.1a** shows the performance evolution of the first (rank 1) and last (rank 500) supercomputer in the Top500 ranking over the years. In 1993, the best supercomputer in the world reached a peak performance of 59.7 gigaflops (10^9 Flop/s) while today's best supercomputer reaches a performance of 143.5 petaflops (10^{15} Flop/s).

The petascale was reached for the first time in 2008 and since then, the race to exascale began. Nowadays, many significant scientific and engineering challenges in both simulation and data analysis already exceed petaflops and are rapidly approaching exaflop-class computing needs.

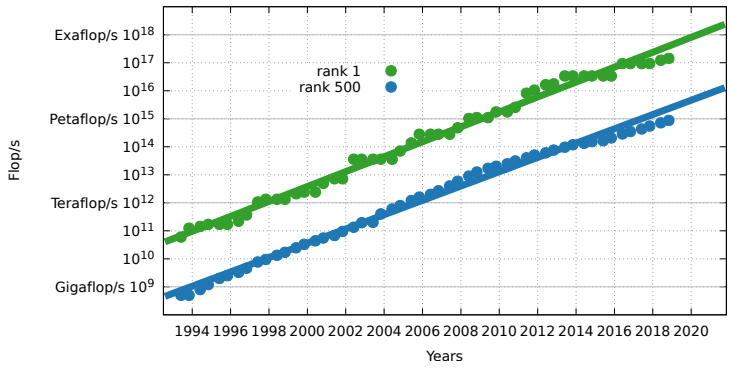
As pointed out by the PRACE Scientific Case for HPC in Europe [1], solving many key science and technology challenges (e.g. Climate science, Nuclear energy, Combustion science, Fusion energy, etc...) will require exascale compute capabilities. For example, in life sciences and medicine, exaflop capabilities will allow the use of more accurate formalisms and enable molecular simulation for high-throughput applications. Molecular simulation is a key tool for computer-aided drug design and appropriate exascale resources could revolutionise this area [1]. In astrophysics, exascale resources will allow simulation of the large-scale universe with sufficient particles to resolve all dark matter halos that could host stars [1].

The projected performance evolution of supercomputers is presented in **Figure 1.1b**.As pointed out on the figure, exascale systems (10^{18} Flop/s) are expected by 2020. However, many challenges need to be addressed to achieve exascale.

[1]The Linpack Benchmark is a measure of a computer's floating-point rate of execution



(a) Performance Evolution.



(b) Performance Evolution Projection.

Figure 1.1: Exponential growth of supercomputing power as recorded by the TOP500 list.

1.1.3 Challenges for Programmability

As emphasized by the International Exascale Software Project Roadmap (IESP) [5], one of the major challenges to achieve exascale is programmability. Programmability is the cross-cutting property that reflects the ease by which application programs may be constructed. It involves several stages of application development: (1) program algorithm capture and representation; (2) program correctness debugging; (3) program performance optimization.

Program algorithm capture and representation The rising complexity of parallel architectures is a major concern for application developers and one of the main challenges to achieve exascale is to improve the programmability of heterogeneous architectures in particular. Hence, new ways of specifying computations are required. Sufficient parallelism must be exposed to maintain exascale operation but with more abstraction of the underlying architecture. The way of specifying parallel algorithm must be decoupled from the complexity of the architecture. For example, scientists must be free from the details of managing data movement between multiple devices and the program representation should be portable across different architectures.

Program correctness debugging An integral part of application development includes verifying that code runs as expected. The large simulation codes today are more and more complex. They incorporate multidiscipline, multi-physics, multiple time scale and multiple solution methods. Cer-

tain codes have taken years to develop and can include millions of lines of code. With the scale and complexity of the science problems enabled by exascale systems, searching manually for a single anomaly among millions of running processes and millions of lines of code is not tenable. Hence, new automatic techniques are required for making sure that the calculations are performed correctly. Application teams specifically request lightweight tools to diagnose memory, threading, and message passing errors that are easy to use and scale with huge code size and high level of parallelism [5].

Program performance optimization Programmability and performance are tightly coupled. For HPC, a major factor affecting programmability has been performance optimization. The gain in performance comes with an increase of application development complexity. Optimizing for a parallel architecture requires now to be an expert in HPC. This relates to the exposure of application parallelism, locality management and load balancing, and memory hierarchy management. These components are expected to be even more important for exascale systems. The complexity at that extreme scale will require that the responsibility for all but parallelism be removed from the programmer. Hence, new tools that can be used by non-expert users are required to automatically handle performance optimization and load balancing issues.

To summarize, new methods are required to assist developers in all stages of application development in order to improve programmability. Application developers should be able to express the parallelism of their application in a portable manner across different devices and architectures (e.g. GPUs, CPUs). The program representation should abstract the details of the underlying architecture and the details of managing data movements between different devices. New methods are required to automatically adapt codes to different architectures and provide some performance portability without degrading the program representation. Moreover, load balancing between different devices must be handled automatically. Finally, new automatic techniques are required to assist application developers during the correctness debugging stage.

The two next sections present the different programming models to express parallelism and introduce the key concepts and state-of-the-art techniques for debugging HPC applications.

1.2 Expressing Parallelism

This section presents the main programming models and languages to express parallelism.

Parallel architectures fall into two broad categories: shared memory and distributed memory. In shared memory architectures a single memory address space is accessible to all processors. This architecture corresponds to common multi-core processors where all cores share the same main memory. In distributed memory architectures each processor owns its own memory. This architecture corresponds to distributed environment such as cluster of computers.

These two types of architectures have given rise to two different approaches to programming parallel systems: shared memory approaches and distributed memory approaches. In shared memory approaches, data are shared among multiple processes executed in parallel and it is the programmer's responsibility to ensure the coherency of concurrent accesses in global memory. On the other hand, in distributed memory approaches, each process owns its own data. Data are moved from the address space of a process to another process address space through cooperative operations on each process. The programmer has to specify what data to send and where.

The two predominant programming models for these approaches are OpenMP for shared memory and MPI for distributed memory. In addition to these programming models, approaches that enable a tradeoff between distributed and shared memory approaches by simulating a global memory space in a distributed environment has led to PGAS programming models. Furthermore, the emergence of heterogeneous architectures has led to parallel programming models devoted for accelerators (OpenACC, CUDA, OpenCL). All these programming models are presented in the next sections.

1.2.1 OpenMP

OpenMP [6] is a shared memory programming model for C, C++ and Fortran. It is based on the fork-join model of parallel execution. An OpenMP program begins as a single thread of execution which creates a team of threads when it encounters a `parallel` construct (fork). At the end of the parallel region, all threads are asleep except the initial thread (join). Different work-sharing constructs can be used to specify how to assign independent work to one or all threads in the team, for example:

- `omp for` is used to split up loop iterations among the threads (loop parallelization).
- `sections` to assign consecutive but independent code blocks to different threads.
- `single` to specify a code block that is executed by only one thread.

In OpenMP, most variables are visible to all threads by default, opening the possibility of data races on a shared variable. However, the programmer can apply data sharing clauses on variables to specify whether they must be shared by all thread or whether each thread must have its own copy of the variable.

Inside a parallel region, synchronization constructs (e.g., `omp barrier`) coordinate threads in a team and data accesses. The specification requires all threads of a team executing a parallel region to execute a barrier or none at all. An implicit barrier is called at the end of `parallel`, `sections`, `single`, and `omp for` constructs unless a nowait clause is specified. Thus, the OpenMP specification forces all threads of a team to encounter the same sequence of work-sharing constructs.

1.2.2 MPI

The Message Passing Interface [7] (MPI) is a standardized interface that allows applications to use message-passing communication. Basically, one process is executed on each node of the cluster and a unique identifier (called rank) is given to each of them. Each MPI process executes a parallel instance of the same program but the control flow of each process diverges according to its rank. Each process owns its private address space and exchanges data across distributed memory systems via messages. MPI exposes multiple ways to exchange data and synchronize between processes, including point-to-point (send / receive) and collective communications. While point-to-point communications involve only two processes, collective communications involve a group (called communicator) of processes. All communications can be blocking or non-blocking.

There are three main types of collective operations: Synchronization (barrier), Data Movement (broadcast, scatter/gather, all to all) and Collective Computation (reductions). **Figure 1.2** shows

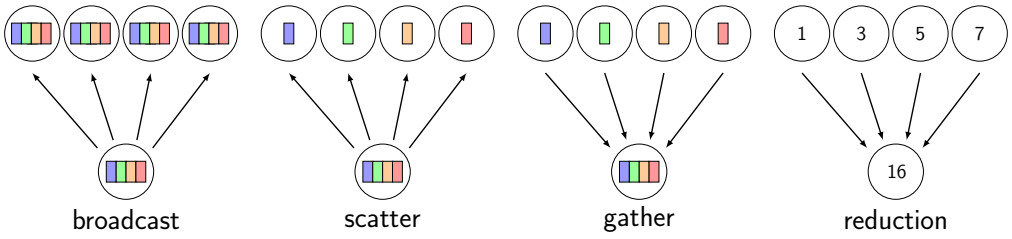


Figure 1.2: MPI Collective Operations.

the four basic collectives operations. `MPI_Bcast` broadcast a message from the process with rank root to all processes of the group, itself included. With the `MPI_Scatter` function, the root process scatters its send buffer to all other processes in the group. Conversely, with the `MPI_Gather` function, each process, root process included, sends the contents of its send buffer to the root process. The `MPI_Reduce` function performs a global reduction operation (e.g. sum, min, max, etc.). One member of the group collects data from the other members and effectuates the reduction operation on that data.

The MPI specification requires that all processes must call blocking and non-blocking collective operations in the exact same order. If all processes in a communicator do not participate in the collective, unexpected behavior, including program failure, can occur.

1.2.3 Partitioned Global Address Space

Partitioned Global Address Space [8] is a distributed shared memory model supporting the notion of shared memory in distributed architectures. It allows a global view of data by an abstracted shared address space and hides the distinction between shared and distributed memory. Each thread owns a portion of a virtually shared memory in addition to its private memory and threads can access the following memories sorted by affinity: local private memory, local shared memory and remote shared memories.

Many languages implement this model. On the one hand, some of them extend existing languages such as Unified Parallel C (UPC) [9] which is based on C, Co-Array Fortran (CAF) [10] which is based on Fortran and Titanium which is based on Java [11]. On the other hand, some of them are new languages dedicated to the model such as Chapel [12].

1.2.4 Programming Many-Core

Existing frameworks for multi-core architectures cannot be used directly on many-core architectures such as GPUs. The complex memory hierarchy dismisses most frameworks for shared-memory architectures. In addition, constraints on IOs (memory allocation, etc.) and memory sizes make models such as MPI unsuitable. Most known and used models devoted to work on many-core architectures are presented below.

OpenACC

OpenACC [13] is a specification defining a set of compiler annotations (pragmas) for C and Fortran programs similar to those of OpenMP. It enables offloading of compute-intensive loops and code regions from a host CPU to an accelerator using simple compiler directives.

OpenCL

OpenCL [14] is a specification that has been endorsed by a large panel of vendors. It provides a common abstraction to program various architectures from different vendors. The OpenCL platform consists in a host processor connected to one or more *compute devices* (e.g. CPUs, GPUs). A compute device consists in one or more *compute units* (CUs) which in turns comprise multiple *processing elements* (PEs). A PE is a virtual scalar processor.

The OpenCL programming model uses a single-instruction multiple-thread (SIMT) model that enables implementation of general-purpose programs on heterogeneous CPU/GPU systems. An OpenCL program consists of a *host program* and *compute kernels*. The host program executes on the host processor and submits commands to perform computations on the devices or to manipulate buffers.

When the host program submits a command to execute a kernel, it defines the parallel iteration space of the kernel as an N-dimensional index space, where $1 \leq N \leq 3$. Each point in the index space is specified by an N-tuple of integers with each dimension starting at 0. Each point is associated with an execution instance of the kernel, which is called work-item (or thread). The N-tuple associated with each thread defines its global ID. These threads are grouped into *work-groups*. Each work-group has a unique ID that is also an N-tuple and each thread inside a work-group has a unique local ID. Hence, each thread has a local ID, a work-group ID and a global ID for each dimension of the NDRange.

The programming language that is used to write compute kernels is called OpenCL C and is based on C99 but adapted to fit the model in OpenCL. In the OpenCL C language, thread identifiers are obtained with the three functions: `get_global_id`, `get_group_id` and `get_local_id`.

When executing a kernel, work-groups are mapped to CUs, and threads are assigned to PEs. Each thread executes the same function, but with different thread identifiers. Threads within a workgroup execute concurrently on the PEs of a single CU and can synchronize and share memory.

OpenCL allows developers to write computation kernels that can run on various device architectures thanks to built-in just-in-time compilation. At runtime the kernels source code is loaded using the function `clCreateProgramWithSources` from the OpenCL API. Then kernels can be compiled for one or more devices using the function `clBuildProgram`. After the program has been compiled, a typical OpenCL application consists in the following steps:

1. First, buffers are allocated on the device, using the `clCreateBuffer` function which takes as parameter the size of the buffer.

2. Then input data are transferred from the host memory to the device memory using the function `clEnqueueWriteBuffer` which takes as parameter a pointer to the host memory region to transfer, the number of bytes to transfer and the buffer into which the data has to be transferred.

3. After the input data have been transferred to the device memory, the sequence of kernels can be executed on the device. Each kernel is launched using the `clEnqueueNDRangeKernel` function which takes as parameter the kernel to execute and the NDRange defining its parallel iteration space.

4. Finally, the output data are transferred from the device memory back to the host using the `clEnqueueReadBuffer` function.

CUDA

CUDA [15] is the programming model developed by NVIDIA on which OpenCL was based. CUDA is similar to OpenCL but is limited to NVIDIA GPUs.

1.2.5 Summary

This section summarizes parallel programming models used in HPC. One major challenge for programmability is the complexity of programming heterogeneous multi-device architectures.

A solution to program these architectures is to combine different programming models (e.g. OpenMP or MPI for CPUs and CUDA for GPUs). This approach is known as hybrid (parallel) programming. However, this forces the developer to write multiple versions (e.g OpenMP and CUDA) for each task/kernel of his program.

To help developer programming heterogeneous architectures, OpenCL provides a common abstraction for different architectures and directly supports multi-GPU and GPU+CPU programming. However, the developer still has the responsibility to adapt the number of tasks to the number of devices, handle the data transfers between the devices and manually balance the load between devices.

Improving programmability of heterogeneous architectures is one of the challenges targeted in the scope of this thesis.

1.3 Debugging HPC Applications

Correctness verification and debugging are important concerns for programmability. Bugs can cost a lot of money. A striking example is the case of the Ariane 5 disaster [16] where a failed numerical conversion resulting in an integer overflow caused the self-destruction mechanism to trigger only 40 seconds after the rocket ignited its engines. The cost of the disastrous launch was estimated to approximately 370 million dollars. Correctness debugging is an even more important concern for HPC applications where it is inconceivable to spend hours of processing time on a supercomputer to compute erroneous results. Over the last decades technologies for verification and debugging have made significant strides is the context of general software development. However, new techniques for verification and debugging of HPC applications are required to achieve exascale as the complexity of algorithms, applications, and architectures are moving beyond the ability of developers. New advances in this domain can lead to substantial improvements in the productivity and sustainability of HPC software development.

This section introduces the key concepts of application correctness debugging in the context of high performance computing.

1.3.1 Definitions

It is important to note that the terminology concerning software problems is not entirely consistent in the literature. The terms “bug”, “error”, “mistake”, “fault”, and “failure” may be used interchangeably and can lead to confusion. More rigorous definitions of these terms from the IEEE Standards [17] are given as follows:

Definition 1. *A mistake is defined as a human action that produces an incorrect result. For example, an incorrect code written by a programmer.*

Definition 2. *A fault (or defect or bug) is an incorrect step, process or data definition in a computer program.*

Definition 3. *An error is defined as a difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.*

Definition 4. *The term failure is used for an observable incorrect program behavior.*

To summarize, a mistake may cause a defect (or bug), and a bug may cause a failure and errors. This leads us to the definition of debugging:

Definition 5. *Debugging is defined as the detection, location and correction of faults in a program.*

1.3.2 Parallel Debugging

Due to concurrency, synchronizations and data exchanges between processes, parallel programs introduce the possibility of new bugs that may not occur in sequential programs (e.g. deadlocks). This section introduces the different types of software bugs that may arise in parallel applications and presents the main challenges for high performance computing.

Parallel bugs classification

The different types of bugs that may arise in parallel applications can be classified into seven categories [18]: *Data Race*, *Deadlock*, *Livelock Starvation*, *Suspension*, *Order Violation*, and *Atomicity Violation*.

A **Data race** occurs when at least two threads or processes access the same data concurrently and at least one write the data. Here "concurrently" means that there wasn't any mechanism that forced one operation to happen before or after the other. This can lead to memory inconsistency if the result of the write operation is not visible to a read operation by another thread (the read operation happens too early). It can also result in a write-write race if a second write operation happens without any read operation in-between.

A **Deadlock** is a program state where each member of a group of threads or processes is waiting for a resource held by another member. The program stays blocked in an infinite waiting state preventing the program to terminate.

A **Livelock** is similar to a deadlock, except that the states of the processes involved constantly changed with regard to one another, none progressing.

Starvation happens when a process is perpetually denied the resources to process its work because other processes are always given preference. It can be caused by scheduling errors or resource leaks for example.

A **Suspension-based locking** occurs when a thread or process waits for an unacceptably long time to acquire a lock for accessing a shared resource.

Order Violation occurs when the desired order between two memory accesses is flipped during execution.

Atomicity Violation happens when the execution on one portion of the code by one process is overlapping with the execution of one portion of the code by another process in such a way that the result is not consistent.

High Performance Computing Challenges

HPC applications have specific characteristics that make traditional testing and debugging techniques much more difficult. We enumerate here three of the most important points.

Non-determinism: Due to high concurrency many HPC programs are non-deterministic. The order of events occurring concurrently on different processors may not always be the same. For example, even with the same input a parallel program does not always have the same behavior from one run to another.

Large Scale: HPC applications are expected to run at massive scale, with very large input size, number of processes and execution time. Some bugs are not observable at smaller scale. This makes HPC programs much more difficult to debug. Testing an application at large scale can take tremendous amount of time and be very expensive. The amount of debugging data to process and the required storage can be huge. Parallel debugging tools are therefore required to scale to that massive scale.

Performance: Expressing algorithms in a natural way and achieving performance are usually two opposite goals. Often, optimization of codes makes them very complex to understand and to debug. Moreover, optimizing a code by hand, which is the case for many programmers, is a very error prone task.

These new challenges advocate for automatic tools and techniques to detect and prevent bugs in HPC applications.

1.3.3 Bug Detection Techniques

Many tools and techniques exist to detect and prevent bugs in HPC applications. These frameworks can be categorized in six groups [19]: *static analysis*, *dynamic analysis*, *formal methods*, *anomaly detection*, *non-determinism control*, and *parallel debugging*. Note that these methods are not mutually exclusive. For example, it is possible to combine the information gathered by a static analysis at compile-time with information obtained at runtime by a dynamic analysis. In the following we give a definition of each of these methods.

Conventional Parallel Debugging

Conventional parallel debugging is the most common used tactic of debugging. Parallel debuggers allow to control and examine the state of threads and processes in a parallel program. Programmers generally start to debug their program from a faulty state and re-launch their program using a debugger in order to explore its state. Among the existing parallel debuggers two of the most popular are DDT [20] and LGDB [21].

Formal Methods

Formal methods allow specification and verification of software. Once a formal specification has been developed, the specification may be used as the basis for proving properties of the specification. For example, the SPIN model checker [22] can be used for verifying the correctness of parallel programming models in a rigorous and mostly automated fashion.

Control of Non-determinism

Many bugs occurring in parallel applications are difficult to reproduce because of non-determinism. To tackle this issue, some tools have been proposed to control the determinism of a parallel application and reproduce bugs more easily. For example, SReplay [23] is a tool for deterministic record and replay for one-sided communications. It allows the user to specify and record the execution of a set of threads of interest (sub-group), and then deterministically replays the execution of the sub-group on a local machine without starting the remaining threads.

Anomaly detection

Anomaly detection is a technique consisting in the identification of behaviors significantly different from the normal behavior of a program in order to isolate potential bugs. For example, the DMTracker tool [24] detects abnormal behaviors in data movements in parallel programs in order to detect potential bugs such as data races and memory corruption.

Static Analysis

Static analysis examines the code without executing the program. The analysis is typically performed at compile-time and warnings are issued for possible errors in a program. For example, the Clang compiler [25] has more than 750 diagnostic flags. Static analyses performed by compilers usually only check the presence of simple errors (e.g. division by zero, null pointer dereference, ...) while more advanced static analyses can reason about the semantic of the program (e.g. find control-flow divergences in an MPI program that may lead to the non-execution of a barrier by all processes). One of the advantages of static analysis is that the overhead of the analysis does not depend on the number of processes as it does not require to execute the program. Moreover, a property that is proven with a static analysis remains true regardless of the input of the program. Nonetheless, the type of properties that can be proven with a static analysis is limited as much information necessary to reason about the correctness of the program is only known at runtime (e.g. input data, execution flow, values of variables). Furthermore, a static analysis may report false positive warnings as it does not take into account execution parameters (e.g. number of threads or

processes) to detect potential bugs in a program. Hence, a warning emitted statically may not be correlated with an actual error at runtime.

Dynamic Analysis

Dynamic analysis checks correctness of the program for a specific input (run). There exists two broad categories of dynamic analyses: online and offline. For online dynamic analyses, checks are performed during program execution while for offline dynamic analyses, checks are performed by analysing traces gathered during application run. With a dynamic analysis, it is much easier to reason about the correctness of a program as runtime information is accessible (e.g. number of barriers encountered by each process of an MPI program). However, as dynamic analysis is specific to a particular run of the application a property that is proven for a given run does not necessarily remain true when the program is executed with a different input set or different execution parameters. Furthermore, a strong limitation of dynamic analysis is that the runtime overhead must remains low and scale with the number of processes.

1.3.4 Summary

This section introduced the key concept of parallel debugging, the challenges due to high performance computing and the main bug detection techniques. In order to achieve exascale, application teams need new tools to help them to debug their codes. A major type of error arising in parallel application are deadlocks and one of the challenges investigated in this thesis is the combination of static and dynamic methods to detect and prevent the apparition of deadlocks in parallel programs.

1.4 Outline and Contributions

This chapter helps to understand the context of this thesis. Scientific applications have an increasing need of resources and many grand scientific challenges require exascale compute capabilities to be addressed. One major concern to achieve exascale is programmability. New automatic methods are required to fill the gap between developers of scientific applications and HPC experts. In addition, as scientific applications are becoming more and more complex and are supposed to run at extreme scale, new tools are required to assist developers in the debugging phase of application development. This thesis explores the combination of static and dynamic methods to improve programmability of HPC applications. Two major issues are investigated: the complexity of programming heterogeneous architectures and the prevention of deadlocks in parallel programs.

The first part of this thesis investigates the automatic task adaptation for heterogeneous architectures. More precisely, we propose a new method to improve programmability of heterogeneous architectures. The programmer expresses the parallelism of his application through a sequence of tasks without considering issues related to the underlying architecture where its code will be executed. Then our method automatically partitions the tasks into sub-tasks executed by each device to take full advantage of the machine capabilities. Our method can adapt to any number of devices and handles load balancing issues stemming from hardware heterogeneity, load imbalance within tasks or between repeated execution of the sequence of tasks. The method is completely transparent

to the user and does not require prior profiling or sampling of the application. The contributions are:

1. The first contribution is the automatic partitioning of irregular tasks to heterogeneous multi-device architectures. (**Chapter 3**);

2. The second contribution is the automatic load balancing of iterated sequences of irregular kernels. (**Chapter 4**);

The second part of this thesis investigates the automatic detection and prevention of deadlocks related to collective operations in parallel programs. Most parallel languages provide collective operations allowing different threads or processes to synchronize or communicate (e.g. `MPI_Broadcast` in MPI, `OMP_Barrier` in OpenMP, `UPC_Barrier` in UPC, `barrier` in OpenCL). A misuse of these collectives can lead to either memory inconsistency or deadlocks. The PARallel COntrol flow Anomaly Checker (PARCOACH) framework combines static code analysis with dynamic instrumentation in order to detect collective misuses in parallel programs and prevent the apparition of deadlocks. In this part of this thesis, we first present the limitations of the PARCOACH debugging method. As PARCOACH analyses each function of a program separately (intra-procedural analysis), we show that in certain situations the feedback reported is inaccurate. We then propose an extension of PARCOACH to overcome these limitations. Finally, we propose to combine the PARCOACH debugging method with a new data-flow analysis in order to compute more precisely concurrent execution paths in parallel programs and reduce the number of false positives returned by PARCOACH. The contributions are:

3. The third contribution of this thesis is the extension of PARCOACH for a fully inter-procedural collective verification. (**Chapter 5**);

4. The last contribution is a new static analysis that detects multi-valued expression in parallel programs (i.e. expressions whose value depends on the rank of processes). We use this analysis to filter out false positives in the PARCOACH tool. We show that our analysis leads to significant improvement over existing debugging methods. (**Chapter 6**).

These contributions have been published and presented in the International European Conference on Parallel and Distributed Computing (Euro-Par) 2016 [26], the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2018 [27], ,the International Workshop on Software Correctness for HPC Applications 2018 [28] and in the International European Conference on Parallel and Distributed Computing (Euro-Par) 2019 [29].

Part I

Automatic Task Adaptation for Heterogeneous Architectures

CHAPTER 2

Automatic Adaptation for Iterated Sequences of Irregular Kernels

3.1 Memory Region Analysis 50
3.1.1 Objectives and Principles 50
3.1.2 Parametric Region Construction and Instantiation 52
3.1.3 Overlapping Write Regions 55
3.1.4 Atomics 57
3.1.5 Limits 57
3.2 Case Study: SOTL 58
3.2.1 Algorithm and Data Structures 59
3.2.2 Kernels Analysis 62
3.2.3 Evaluation 69
3.3 Related Work 70
3.4 Summary 72

Graphic Processor Units (GPU) are ubiquitous and nowadays most computing nodes of a parallel machine consist in GPUs and multicore CPUs. The complexity of these architectures and their programming model adds an additional burden on application developers and one of the major challenges for the race to exascale is to ease the programmability of these architectures.

In terms of programming language, OpenCL has emerged as the programming language for heterogeneous computing, able to define code for GPUs and CPUs alike. However, this introduces new challenges: The application code has to be adapted to the number of devices, the codes of the kernels have to be optimized for each different device, and the workload has to be balanced equally between these devices. Load balancing is difficult to achieve in general, because the architecture is heterogeneous, the parallel application may not have a constant load during its execution and both computation and communication times have to be taken into account.

In recent years, many works have focused on how to improve portability and maximize performance of applications on heterogeneous multicore and multi-GPU platforms for OpenCL, OpenMP or CUDA (see [30–35] to name a few). These runtimes implement different scheduling strategies, but there is no task partitioning involved. It is assumed the developer has created a task graph with

enough parallelism to feed the different devices and ensure a possible load balancing among the different units. Some works propose their own API [30, 36], a domain specific language [37], or a set of compiler directives [OpenACC, OpenMP4.0] to advise the compiler and runtime to offload a block of code to an accelerator. Other works automatically generate OpenCL code from a data-parallel shared memory program written in OpenMP [38], or generate a CUDA code from an OpenMP program extended with a new set of compiler directives and environment variables [39].

More recent works propose to adapt parallelism to the number of devices, by partitioning the kernels. Few of them tried to leverage irregular applications (for instance [40–44]). This is particularly challenging for real applications, where multiple kernels are executed, with possible load variations within one kernel and heterogeneous devices.

In this part of this thesis, we focus on applications with an iterated sequence of kernels. This occurs in iterative computations, for instance until a fixed point is reached or for a simulation, where each iteration corresponds to a time step. This chapter presents the context and parallelization model we propose to leverage the compute capabilities of heterogeneous multi-device architectures, the main challenges we need to address and the principle of our method to automatically adapt single-device applications to heterogeneous multi-device architectures.

2.1 Context and Parallelization Model

This section presents the parallelism model studied in this thesis, the type of targeted applications as well as the parallelization model proposed to take advantage of the computing power of multiple heterogeneous devices.

2.1.1 Context

The parallelism we consider is expressed as a parallel loop over a range $\mathcal{R}$. The pseudocode of a parallel computation kernel taking as input an array `A` and as output an array `B` is shown in **Figure 2.1**. Each thread $t \in \mathcal{R}$ executes the same function in parallel but with a different parameter corresponding to its index in the range. Each thread t computes a partial region of the output buffer `B` and requires a partial region of the input buffer `A`. The partial region of `A` required by a thread t is denoted as $f(t)$ and the partial region of `B` computed by t is denoted as $g(t)$.

```
1 kernel(R, A) → B
2 |  parallel for t ∈ R do
3 |  |  B[g(t)] = h(t, A[f(t)])
4 |  end
5 end
```

Figure 2.1: Pseudocode of a parallel computation kernel.

The applications targeted in this part of the thesis consist in iterated sequences of kernels. The pseudocode of an application with an iterated sequence of m kernels is shown in **Figure 2.2**. For each iteration of the application, the same sequence of kernels is executed. Each kernel k from the sequence is executed over a parallel iteration space defined by its range $\mathcal{R}_k$ and takes as input an array $\mathtt{A}_k$ and as output an array $\mathtt{B}_k$.

The statement line 3 is a simplified notation of the pseudocode shown in **Figure 2.1**. For each kernel k, each thread t in the range $\mathcal{R}_k$ is executed in parallel, the region of the input array $\mathtt{A}_k$ read by kernel k is denoted as $f_k(\mathcal{R}_k)$ and the region of the output array $\mathtt{B}_k$ written by kernel k is denoted as $g_k(\mathcal{R}_k)$.

There can be any data dependency between kernels. For example for two kernels k and k', the output buffer $\mathtt{B}_k$ of kernel k can denote the same buffer as the input buffer $\mathtt{A}_{k'}$ of kernel k'.

```
1 for i = 0; i < iter; i = i + 1 do                    /* for each iteration */
2 |   for k = 0; k < m; k = k + 1 do                      /* for each kernel */
3 |   |   B_k[g_k(R_k)] = kernel_k(R_k, A_k[f_k(R_k)])  /* execution of kernel k */
4 |   end
5 end
```

Figure 2.2: Pseudocode of an application with an iterated sequence of m kernels.

The workload of the application can be irregular and dynamic. An irregular workload means that for a kernel k at iteration i and two different threads $t_1 \neq t_2 \in \mathcal{R}_k$, the execution time of t_1 can be different from the execution time of t_2. A dynamic workload means that given a kernel k and a thread $t \in \mathcal{R}_k$, the execution time of t at iteration i is not necessarily the same as the execution time of t at iteration $i + 1$.

This parallelism model corresponds to many programming languages and parallel libraries. **Figure 2.3** shows the code of an application with an iterated sequence of m kernels where one of them is a stencil for two languages: OpenCL and CUDA; and for one library: the Intel Thread Building Blocks (TBB) library.

The code of the application for OpenCL and CUDA is shown in **Figures 2.3a and 2.3b**. Note that since OpenCL and CUDA may target devices that cannot directly access the physical memory of the CPU (e.g. discrete GPUs), the first step consists in transferring the input data from the host to the device (`clEnqueueWriteBuffer` in OpenCL and `cudaMemcpy` in CUDA). Then for each iteration, each kernel from the sequence is executed on a device. The parallel iteration space of each kernel corresponds to an index space in 1, 2 or 3 dimensions, called *NDRange* in OpenCL and *Grid* in CUDA. Each dimension is split into work-groups (thread blocks in the CUDA terminology). In OpenCL, the NDRange is defined by its *global work size* (`global[k]`) and its *local work size* (`local[k]`). The global work size corresponds to the number of threads in each dimension and the local work size corresponds to the work-group size in each dimension. In CUDA, the Grid is defined by its number of thread blocks in each dimension (`nBlocks[k]`) and by the size of a thread block in each dimension (`blockSize[k]`). The example of a stencil kernel is shown for both languages. Each thread executes the same function but with different indices corresponding to its position in the parallel iteration space. The indices of a thread are given by `get_global_id(0)`, `get_global_id(1)` and `get_global_id(2)` in OpenCL and by `threadIdx.x`, `threadIdx.y`, and `threadIdx.z` in CUDA. Note that there is no need for data transfers between kernels since they are all executed on the same device. However, after the last iteration, the output data are transferred back to the host.

The code of the same application using the Intel TBB library is shown in **Figure 2.3c**. The iteration space of each kernel is defined line 15 and goes from 0 to *range*[k]. The template function `tbb::parallel_for` breaks this iteration space into chunks, and runs each chunk on a separate thread. The code of a 1-dimensional stencil kernel is shown line 4 and is defined by the C++

```
1  // host.c
2  clEnqueueWriteBuffer(...);
3
4  for (int i=0: i<iter; i++) {
5    for (int k=0; k<nbKernels; k++) {
6      clEnqueueNDRangeKernel(kernel[k],
7                    global[k],local[k]);
8    }
9  }
10
11 clEnqueueReadBuffer(...);
12
13 // device.cl
14 __kernel void
15 stencil(__global float *A,
16         __global float *B,
17         int n) {
18   int i = get_global_id(0);
19   if (i < 1 || i > n-1)
20     return;
21   B[i] = 0.3 * (A[i-1] + A[i] + A[i+1]);
22 }
```

(a) OpenCL.

```
1  // host.c
2  cudaMemcpy(..., cudaMemcpyHostToDevice);
3
4  for (int i=0: i<iter; i++) {
5    for (int k=0; k<nbKernels; k++) {
6      kernel[k] <<< nBlocks[k],
7                    blockSize[k] >>> (...);
8    }
9  }
10
11 cudaMemcpy(..., cudaMemcpyDeviceToHost);
12
13 // device.cu
14 __global__ void
15 stencil(float *A, float *B, int n)
16 {
17   int i = threadIdx.x;
18   if (i < 1 || i > n-1)
19     return;
20   B[i] = 0.3 * (A[i-1] + A[i] + A[i+1]);
21 }
```

(b) CUDA.

```
1  // stencil.cpp
2  class Stencil {
3    ...
4    void operator()(const blocked_range<size_t>& r ) const {
5      for(size_t i=r.begin(); i!=r.end(); ++i ) {
6        if (i >= 1 && i <= n-1)
7          B[i] = 0.3 * (A[i-1] + A[i] + A[i+1]);
8    }
9    ...
10 };
11
12 // main.cpp
13 for (int i=0; i<iter; i++) {
14   for (int k=0; k<nbKernels; k++) {
15     parallel_for(blocked_range<size_t>(0,range[k]), kernel[k](...));
16   }
17 }
```

(c) Intel TBB.

Figure 2.3: Single device iterative application with m kernels written in different languages.

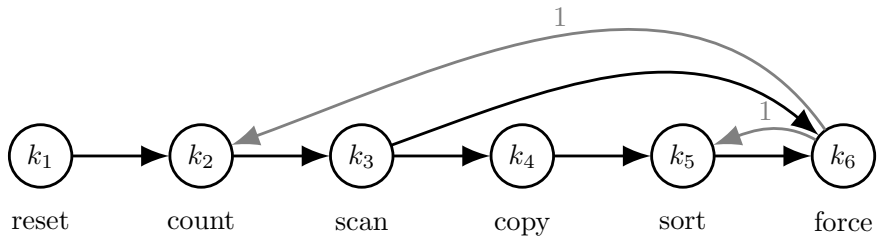
operator `operator()`. Note that since the computation kernels are executed on the CPU, no data transfer is required.

Transforming by hand a single-device application into a multi-device application that leverage the computational power of multiple devices is very complex and error prone. The number of kernels need to be adapted to the number of devices and the burden of managing the data transfers and balancing the load between the devices is left to the programmer. In addition, each time the code is executed on a new machine, it must be adapted again. The challenge we want to address is: Given a single-device iterative application with m kernels and n heterogeneous devices (GPUs, CPUs), how to automatically adapt the application in order to minimize the total execution time.

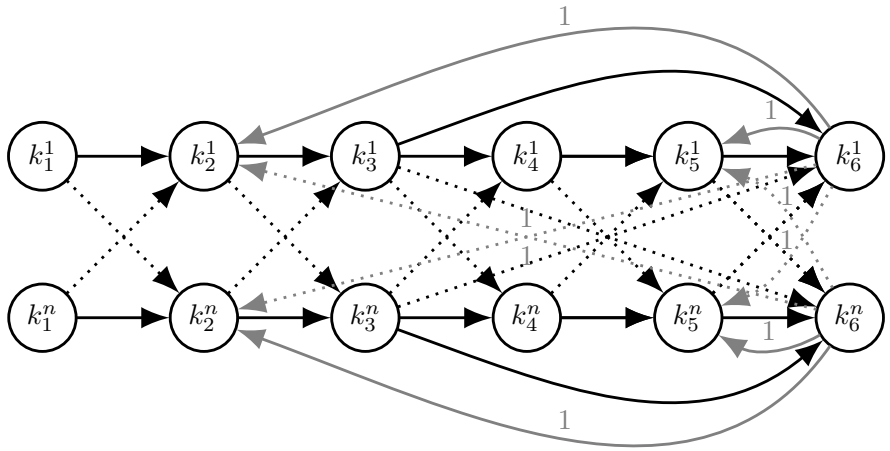
2.1.2 Parallelization Model

Let us consider a real multi-kernel iterative application. SOTL is an N-Body simulation consisting in an iterated sequence of six kernels where each iteration corresponds to a time step. **Figure 2.4a** presents the different kernels of SOTL with their names. The purpose of each kernel will be presented in **Section 3.2**. The kernels are repeatedly executed in a loop and edges show the dependences among these kernels, with their dependence distance. 1 means that the target kernel depends on the result computed by the source kernel one iteration before. We can see that there is not enough parallelism to schedule the kernels onto multiple devices.

The parallelization model we propose is to distribute the parallel iteration space of each kernel onto the devices. The parallelization of SOTL onto *n* devices is shown in **Figure 2.4b**. Each kernel is split into several sub-kernels, one per device. We define a sub-kernel of a kernel *k* as a kernel only executing a slice of the original range of kernel *k*. The sub-kernel of a kernel *k* executed on device *d* is denoted ad k_k^d in **Figure 2.4b**. Splitting each kernel from the sequence into several sub-kernels may imply data dependencies between sub-kernels executed on different devices. Such dependencies are shown with dashed edges on the figure. When such dependencies occur, data transfers may be required if the devices do not share the same physical memory.



(a) Dependence graph between kernels.



(b) Parallelization.

Figure 2.4: Parallelization of the SOTL application.

The pseudocode on **Figure 2.5b** shows in more details how an application with *m* kernels may be adapted to leverage the computational power of *n* devices. First, a partitioning of each kernel

has to be computed, this is the role of the statement line 3. The function $split(\mathcal{R}_k)$ splits the iteration space of a kernel k defined by its range $\mathcal{R}_k$ into several sub-ranges $\mathcal{R}_k^d$ such that $\bigcup_{1 \leq d \leq n} \mathcal{R}_k^d = \mathcal{R}_k$ when n is the number of devices. Then the input and output arrays must be split, this the role of the function *dsplit* in the statements lines 4 and 5. When the different devices do not share the same physical memory (e.g. discrete GPUs), each device will need its own input buffer to read the partial region of the input array required by its sub-kernel and its own output buffer to write the partial region of the output array it computes. The way these arrays will be split may depend on the sub-range of the kernel executed onto each device. For that reason, the function *dsplit* takes as parameter the sub-ranges $\mathcal{R}_k^1, \ldots, \mathcal{R}_k^n$ of each device. After the data have been split, each device will execute in parallel a sub-kernel, that is the same kernel but only for threads belonging to its sub-range. This is the role of the statements line 6 and 7. The region of the input array read by the sub-kernel of kernel k executed on device d is denoted as $f_k(\mathcal{R}_k^d)$ and the written region of the output array is denoted as $g_k(\mathcal{R}_k^d)$. Finally, the partial results computed onto each device have to be merged, this is the role the function *dmerge* in the statement line 9.

```
1  for i = 0; i < iter; i = i + 1 do                    /* for each iteration */
2  |  for k = 0; k < m; k = k + 1 do                     /* for each kernel */
3  |  |  B_k[g_k(R_k)] = kernel_k(R_k, A_k[f_k(R_k)])   /* execution of kernel k */
4  |  end
5  end
```

(a) Single-device iterative application with m kernels.

```
1  for i = 0; i < iter; i = i + 1 do                    /* for each iteration */
2  |  for k = 0; k < m; k = k + 1 do                     /* for each kernel */
3  |  |  R_k^1 ... R_k^n ← split(R_k)
4  |  |  A_k^1 ... A_k^n ← dsplit(A_k, R_k^1, ..., R_k^n)
5  |  |  B_k^1 ... B_k^n ← dsplit(B_k, R_k^1, ..., R_k^n)
6  |  |  parallel for d = 1; d ≤ n; d = d + 1 do       /* for all devices */
7  |  |  |  B_k^d[g_k(R_k^d)] = kernel_k(R_k^d, A_k^d[f_k(R_k^d)])
8  |  |  end
9  |  |  B_k ← dmerge(B_k^1 ... B_k^n)
10 |  end
11 end
```

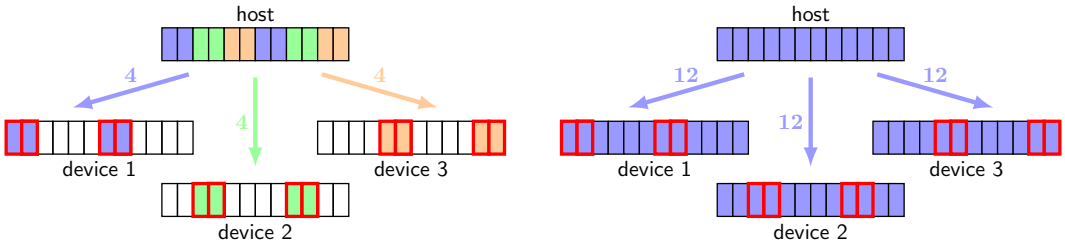
(b) Parallelization onto n devices.

Figure 2.5: Transformation of a single-device iterative application into a multi-device application.

Now that we have introduced the parallelization model to leverage the compute capabilities of multiple devices, the problem is how to implement the functions *split*, *dsplit* and *dmerge*. In other words, how to split the computation of each kernel in order to minimize the total execution time of the application and how to merge the partial results computed by different devices.

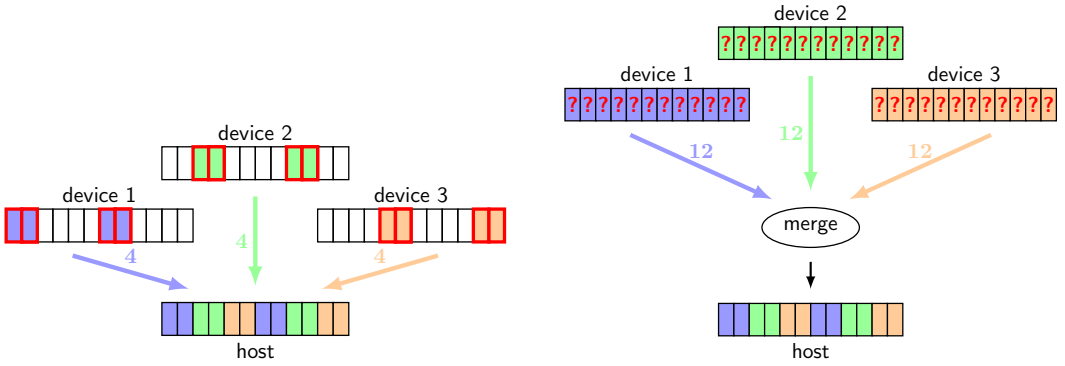
2.1.3 Partitioning Strategies

Several strategies are possible to split the data and merge the partial output from different devices (*dsplit* and *dmerge* functions). In order to split input arrays, an easy solution is to broadcast the whole array to every device. However, this solution implies unnecessary data transfers. Conversely, a precise knowledge of the data required by each thread allows to only transfer the required data to each device and results in shorter communication time. **Figures 2.6a and 2.6b** illustrate these two strategies. The region of the input array required onto each device is highlighted in red. With a precise knowledge of the required data, only 4 elements are transferred from the host to each device instead of 12 when the whole array is broadcasted.



(a) Memory Region Analysis. **(b)** Broadcast.

Figure 2.6: Partitioning Input Arrays. The real region required onto each device is highlighted in red. With a memory region analysis, only the required data is transferred from the host to each device.



(a) Memory Region Analysis. **(b)** Merge.

Figure 2.7: Partitioning Output Arrays. The partial region written by each device is highlighted in red. With a memory analysis only the partial regions written by each device are transferred to the host.

For output arrays, finding how to split the data is essential to merge the partial modification computed by each device. When the written region of each device is precisely known, bringing back the data to the host can be done in parallel. On the contrary, if the partial region of the output buffer written by each device is unknown, a merging operation is necessary to build the output array. This merging operation consists in comparing the output arrays from each device with the value of the output array before sub-kernels execution in order to merge the modification from every device. **Figures 2.7a and 2.7b** illustrate these two situations. The regions of the output array written by

each device are highlighted in red. With a precise knowledge of the data written, only 4 elements are transferred from each device back to the host in **Figure 2.7a**. Conversely, in **Figure 2.7b**, when the regions written by each device are not known statically, the whole array from each device has to be transferred back to the host, then a "merge" kernel has to be executed to properly merge the modification of each device.

For many applications the overhead induced by these data transfers and by the merging operation may result in a slowdown compared to executing the application on a single device. This is especially the case for multi-kernel applications when there are data dependencies between kernels.

Let's consider the case where there is a data dependency between two kernels. The pseudocode on **Figure 2.8a** shows an example where the output array `B` computed by a kernel k serves as input array for a kernel k' executed after k. When partitioning these kernels onto multiple devices, the partial results of array `B` computed by each device must be merged after the execution of the sub-kernels of kernel k. Then, the array `B` must be split again before the execution of the sub-kernels of kernel k'. This is the role of the two statements lines 5 and 6 in **Figure 2.8b**.

However, when for each device, the region of array `B` required by kernel k' has been computed by the sub-kernel of kernel k executed on the same device, these merge and split operations can be avoided. Indeed, the data required by each sub-kernel of kernel k' is already present in the memory of the device where it is executed. Hence, no data transfer is required.

In addition, in some cases a possible optimization opportunity would be to merge the two parallel for loops lines 2 and 7. Nevertheless, this opportunity of optimization is not always possible as the partial region of `B` written by each sub-kernel of k and the partial region of `B` read by each sub-kernel of k' depend on their sub-ranges and their sub-ranges depends on the load balancing.

```
1 ...
2 B[g(R)] = kernel(R, A[f(R)])
3 C[g'(R')] = kernel'(R', B[f'(R')])
4 ...
```

(a) single device.

```
1 ...
2 parallel for d = 0; d < n; d = d + 1 do
3 |   B^d[g(R^d)] = kernel(R^d, A^d[f(R^d)])
4 end
5 B ← dmerge(B^1 ... B^n)
6 B^1 ... B^n ← dsplit(B, R'^1, ..., R'^n))
7 parallel for d = 0; d < n; d = d + 1 do
8 |   C^d[g'(R'^d)] =
  |     kernel'(R'^d, B_d[f'(R'^d)])
9 end
10 ...
```

(b) multi-device.

Figure 2.8: Splitting data when there is a data dependency between two kernels.

Determining the region of input and output arrays read and written by each sub-kernel is even more important for iterative applications. For stencil applications for instance, transferring cells from the whole domain from each device back to the host, executing a "merge" kernel and then transferring the merged array again from the host to each device for the next iteration can ruin performance. On the other hand, with a precise knowledge of the memory region accessed by each device only cells from the border of the domain are exchanged after each iteration. **Figure 2.9**

illustrates the communications generated with a precise knowledge of the memory regions accessed by each device when a 1-dimensional stencil of size 12 is partitioned on two devices. At each iteration, each thread computes a new value for the cell corresponding to its position in the parallel iteration space from the value of the cell at the same position in the input array as well as the values of the two neighboring cells and writes it into the output array. For the first iteration, array `A` is used as input to compute the new stencil values into array `B`. Then the input and output arrays are switched at each iteration. Before the first iteration, only the data required onto each device is transferred from the host to each device. Then the sub-kernels are executed and each device computes one half of the values of array `B`. For the next iteration, `B` is used as input array and `A` as output array. Since all elements of `B` required by each sub-kernel except one have been computed on the same device, only one element from each device is transferred between the two iterations. Finally, after the last iteration only the data computed by each device is transferred back to the host.

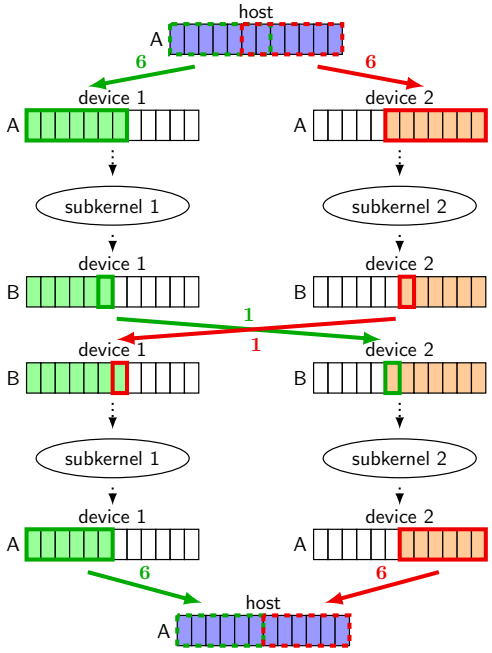


Figure 2.9: Stencil Partitioning.

This section shows the importance of precisely determining the region of input and output arrays read and written by each sub-kernel. A precise memory region analysis can significantly reduce the volume of communications between devices.

2.2 Challenges

To automatically adapt a single-device application to multiple heterogeneous devices, the problem is twofold.

First it is necessary to know for each kernel how to partition the data depending on the slice of the original iteration space executed onto each device. That is the same as calculating for each kernel k and for each array `B` the functions $f_k^{\mathtt{B}}(\mathcal{R})$ and $g_k^{\mathtt{B}}(\mathcal{R})$ giving for any range $\mathcal{R}$ the partial region required and written by all threads in $\mathcal{R}$.

Second, the load must be balanced dynamically between the devices. We need to determine for each iteration of the computation, the decomposition of the iteration space $\mathcal{R}_k$ of each kernel k into sub-ranges $\mathcal{R}_k^d$ minimizing the total execution time of the application.

The problem of data partitioning corresponds to the implementation of the *dsplit* and *dmerge* functions presented in **Figure 2.5b, page 30** while the problem of load balancing corresponds to the implementation of the *split* function.

The main challenges for partitioning the data and balancing the load are presented in sections **Section 2.2.1** and **Section 2.2.2** by studying OpenCL applications.

2.2.1 Data Partitioning

We illustrate the challenges of partitioning data on three OpenCL kernels: a 1-dimensional stencil, a 2-dimensional stencil and a kernel with indirect memory access.

1D Stencil

Let's consider the 1-dimensional stencil kernel shown in **Figure 2.10**. At each iteration the values of the cells at the previous iteration are read from buffer `A` to compute the new value of each cell into buffer `B`. The parallel iteration space of the kernel defined by its NDRange corresponds to the number of cells in the stencil, and each thread computes in parallel a new value for the cell corresponding to its index in the NDRange.

```
1 void stencil1D(float *A, float *B, int n)
2 {
3   int i = get_global_id(0);
4
5   if (i >= 1 && i <= n-1) {
6     B[i] = 0.3 * (A[i-1] + A[i] + A[i+1]);
7   }
8 }
```

Figure 2.10: 1D Stencil kernel.

A static analysis of the code allows to detect each load/store operation from/into the buffers given as parameters of the kernel. This kernel takes as parameters two buffers: `A` and `B`. There is three load operations (`A[i-1], A[i], A[i+1]`) and one store operation (`B[i]`) corresponding to the statement at line 6. Following the def/use chain [45] it is possible to recursively replace each value with its definition to determine the indices of the elements read or written by each memory access. Here `i` is replaced with `get_global_id(0)`. Each thread reads from buffer `A` the elements at indices:

`{get_global_id(0)-1,get_global_id(0),get_global_id(0)+1}`

and writes one element into buffer `B` at index:

`{get_global_id(0)}`.

Note that the statement line 6 is only executed if the condition line 5 is satisfied. After replacing each value with its definition, this condition can be computed as:

```
get_global_id(0) >= 1 and get_global_id(0) < n
```

with `n` a scalar parameter of the kernel.

To compute the read and written regions of each buffer, one has to replace `get_global_id(0)` and `n` with their values taking into account the condition line 5. With an interval analysis, it is possible to compute these regions by replacing `get_global_id(0)` with the interval corresponding to the slice of the NDRange executed onto each device. However, the NDRange of the kernel, the slice executed onto each device and the value of the scalar parameter `n` are only known at runtime.

This example shows that a combination of static and dynamic analyses is necessary to partition the data of a kernel automatically. A static analysis of the code can compute parametric regions of the buffers accessed by the kernel but these regions may depend on parameters that are only known at runtime such as the NDRange of the kernel or the values of scalar parameters.

2D Stencil

The code of a 2D stencil is shown in **Figure 2.11**. The NDRange defining the parallel iteration space of the kernel is in 2 dimensions. The index of each thread in the first dimension is given by `get_global_id(0)` and the index in the second dimension by `get_global_id(1)`. At each iteration, each thread computes a new value for the cell whose row number is its index in the first dimension and column number is its index in the second dimension. Hence, each thread writes into buffer `B` at index `get_global_id(0) * m + get_global_id(1)` where `m` and `n` correspond to the height and the width of the 2D domain.

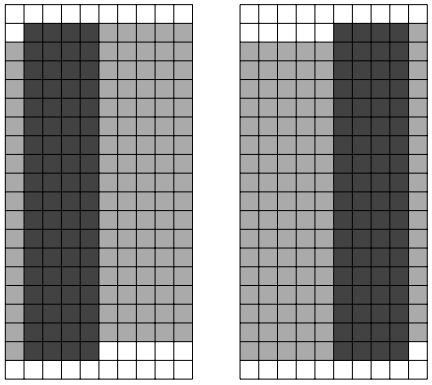
```
1  void stencil2D(float* A, float* B, int m, int n)
2  {
3    int i = get_global_id(0);
4    int j = get_global_id(1);
5
6    if ((i >= 1) && (i < (m-1)) && (j >= 1) && (j < (n-1))) {
7      B[i*n + j] = 0.2f * (A[i     * n + j] +
8                           A[i     * n + (j-1)] +
9                           A[i     * n + (j+1)] +
10                          A[(i+1) * n + j] +
11                          A[(i-1) * n + j]);
12   }
13 }
```

Figure 2.11: 2D Stencil kernel.

Assuming that the size of the domain is 20×10 (`m`=20, `n`=10), the NDRange defining the thread indices in each dimension corresponds to the following intervals $< [0, 19], [0, 9] >$. When partitioning this kernel into sub-kernels, the parallel iteration space can be distributed to different devices by splitting one of the two dimensions of the NDRange.

Let's consider the case where the kernel is partitioned on two devices and each device executes half of the parallel iteration space. If the second dimension is split, the NDRanges are

$< [0, 19], [0, 4] >$ and $< [0, 19], [5, 9] >$ for the first and second device respectively. After evaluating the condition line 6 we can see that only threads whose indices are in $< [1, 18], [1, 4] >$ for the first device and in $< [1, 18], [5, 8] >$ for the second write into buffer `B`. Hence, we can replace the calls to `get_global_id(0)` and `get_global_id(1)` with the intervals of thread indices executing the statement line 7 and `n` with its value. Then using an interval analysis we can compute an overapproximation of the region of buffer `B` written on each device. This gives us $[1, 18] * 10 + [1, 4] = [11, 184]$ and $[1, 18] * 10 + [5, 8] = [15, 188]$ for the first and second device respectively. We can see that the two regions obtained are not disjoint. In fact each thread writes buffer `B` at different indices and the regions written onto each device are disjoint as shown **Figure 2.12**. However, with an interval analysis the rectangular region of buffer `B` written onto each device is overapproximated to a single interval, resulting in two overlapping regions. In this case the device on which the elements of `B` in $[15, 184]$ are written is unknown. Therefore, when splitting the second dimension of the NDRange for this kernel, a merging operation is needed to properly merge the modifications from both devices.

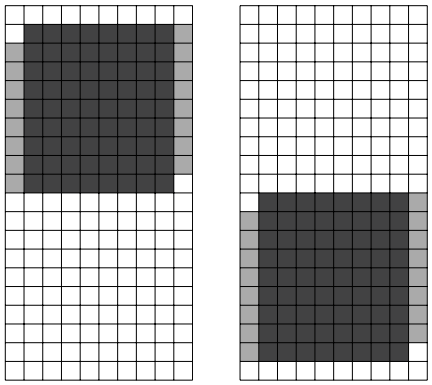


(a) sub-kernel 1 $[11, 184]$. **(b)** sub-kernel 2 $[15, 188]$.

Figure 2.12: Memory regions of buffer `B` written by each sub-kernel when the stencil2D kernel is partitioned on 2 devices by splitting the second dimension of the NDRange. The gray area shows the overapproximation obtained by an interval analysis but in fact only elements in darker gray are actually written by each sub-kernel.

On the other hand, if the first dimension of the NDRange is split, the written regions of buffer `B` computed with an interval analysis are disjoint as shown in **Figure 2.13**. The NDRanges are $< [0, 9], [0, 9] >$ and $< [10, 19], [0, 9] >$ for the first and second device respectively. After evaluating the condition line 6 we can see that only threads whose indices are in $< [1, 9], [1, 8] >$ for the first device and in $< [10, 18], [1, 8] >$ for the second will write into buffer `B`. This gives us $[1, 9] * 10 + [1, 8] = [11, 98]$ and $[10, 18] * 10 + [1, 8] = [100, 188]$ for the first and second device respectively. In this case no merging operation is required.

This example shows that when partitioning a kernel whose parallel iteration space has more than one dimension, the choice of the dimension to split can have a great impact on the amount of data to transfer. In order to avoid a merging operation, the right dimension must be chosen.



(a) sub-kernel 1 [11, 98].

(b) sub-kernel 2 [100, 188].

Figure 2.13: Memory regions of buffer `B` written by each sub-kernel when the stencil2D kernel is partitioned on 2 devices by splitting the first dimension of the NDRange. The gray area shows the overapproximation obtained by an interval analysis but in fact only elements in darker gray are actually written by each sub-kernel.

Indirect Memory Access

The code of a kernel with indirect memory access is shown in **Figure 2.14**. In this kernel, the input buffer `A` is indirectly accessed through the indirection array `IA`. Each thread reads the region of the input buffer `A` corresponding to the interval [`start`, `end` − 1] where `start` is the value of buffer `IA` at the position corresponding to its index in NDRange (`get_global_id(0)`) and `end` is the value of buffer `IA` at the next position.

```
1  void SpatialBinning(float *A, int *IA, float *B) {
2    int id = get_global_id(0);
3    float sum = 0;
4
5    int start = IA[id];
6    int end   = IA[id+1];
7
8    for (int i = start; i < end; i++)
9      sum += A[i];
10
11   B[id] = sum;
12 }
```

Figure 2.14: Spatial Binning kernel.

With a static analysis of the code one can determine that for a thread of index i the region read from buffer `A` is defined by the interval [`IA`[i], `IA`[$i+1$] − 1]. Hence, the region of buffer `A` read by a sub-kernel with sub-range defined by the interval [L, U] is:

$$\bigcup_{i=L}^{U} [\texttt{IA}[i], \texttt{IA}[i+1] - 1].$$

Computing the intervals $[\texttt{IA}[i], \texttt{IA}[i+1]-1]$ for each $i \in [L, U]$ may not be acceptable to determine the region of buffer `A` read by a sub-kernel since the interval $[L, U]$ may be very large. Moreover, computing this union would require to know the values of all elements of buffer `IA` whose positions are within the interval $[L, U+1]$. As buffers are allocated on the devices memory, these elements are not directly accessible from the host. Transferring the region $[L, U+1]$ from buffer `IA` back to the host may induce a significant overhead.

However, assuming that values inside buffer `IA` are increasing, this union of intervals can be approximated to the single interval $[\texttt{IA}[L], \texttt{IA}[U+1]-1]$. In this case, determining the region of buffer `A` read by the sub-kernel would only require to read the two elements of buffer `IA` at indices L and $U+1$. Nevertheless, this would require to ensure that values inside buffer `IA` are increasing without having to read the whole buffer.

This type of indirect memory access where values inside the indirection array are increasing occurs in many applications. For example, it corresponds to spatial data structures where data are sorted into spatial bins. Spatial binning [46] is a key technique for applications such as collision detection or particle simulation.

This example shows the difficulty of partitioning a kernel in the presence of indirect memory accesses. When the values in the indirection array are monotonous, it may be possible to read only two elements from the indirection array to compute the region accessed by a sub-kernel. However, it would require a hint from the user to ensure that the values in the indirection array are monotonous.

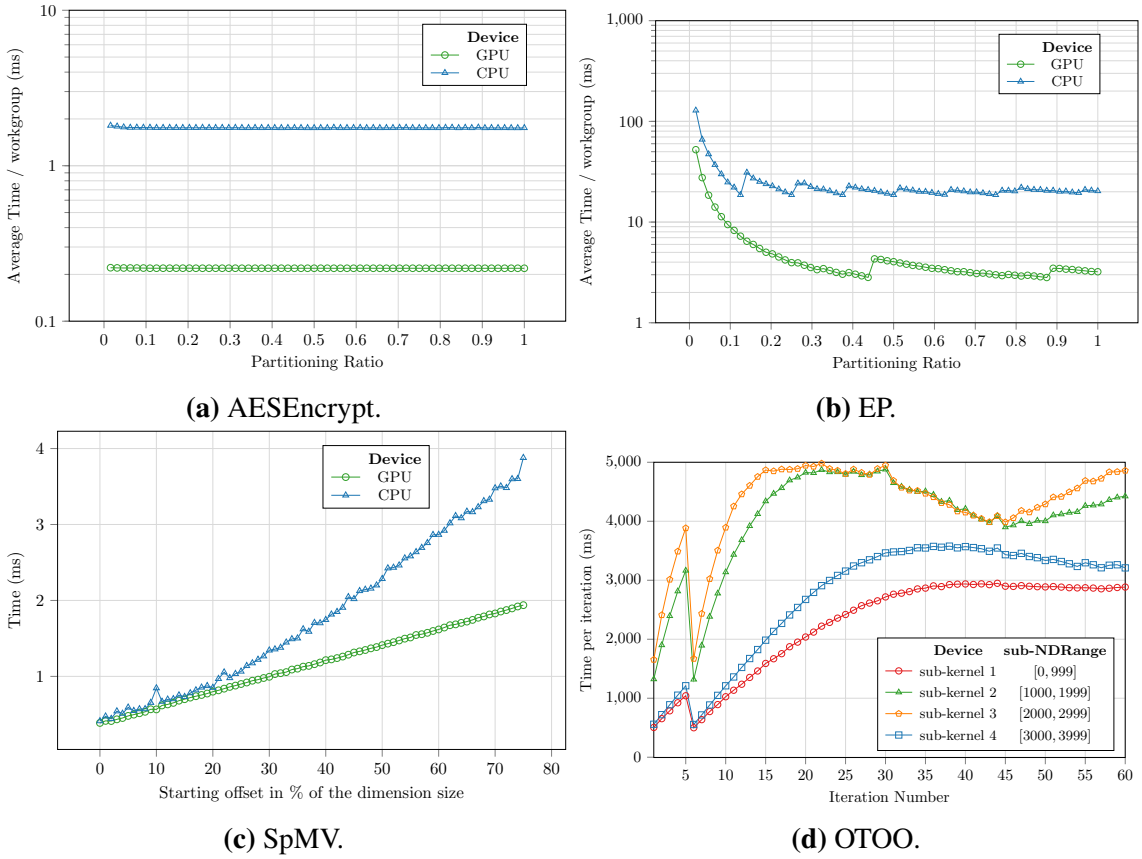
2.2.2 Load Balancing

In this section we evaluate the factors that must be considered to balance the workload between the devices and minimize the total execution time of an application consisting of an iterated sequence of kernels.

As OpenCL kernel executions are characterized by the number of parallel work-groups, we define the *partitioning ratio* of each sub-kernel as the ratio of work-groups it executes over the total number of work-groups of the kernel, a partitioning ratio of 1 meaning the whole kernel is executed.

Impact of architectural heterogeneity

We study the performance variation of a kernel on one device, decreasing manually its partitioning ratio. **Figures 2.15a and 2.15b** respectively show performance of AESEncrypt and EP from SNU NPB Suite [47] for different partitioning ratios on a 16-core Intel Xeon E5-2650 2.00GHz with 64GB (CPU) and on an NVIDIA Tesla M2075 (GPU). Performance is indicated as the mean time to execute one work-group (lower is better). For AESEncrypt, the average time per work-group is nearly constant for all partitioning ratios, and very different on CPU and on GPU. For EP, we observe large performance drops (higher average time/work-group) at regular intervals of partitioning ratios on both CPU and GPU. This may come from compiler optimizations (such as unrolling), cache effects, and inefficient occupancy of the parallel resources due to a low number of work-groups within sub-kernels.



(a) AESEncrypt. **(b)** EP. **(c)** SpMV. **(d)** OTOO.

Figure 2.15: (a) and (b): Impact on performance of architectural heterogeneity on AESEncrypt and EP benchmarks. Performance is given as an average time per work-group, partitioning ratio as a percentage of the total number of work-groups. (c) Impact on performance of the offset (starting index) for SpMV kernel, with a fixed partitioning ratio of 1/4. (d) Impact of iteration number on performance for OTOO application. Partitioning ratios are set to 1/4 for all GPUs, and offset is fixed on all devices.

Impact of the work-group offset

Work-items (or threads) are indexed in OpenCL by a vector of indices among a rectangular space (from 1D to 3D) called the NDRange. Selecting a partitioning ratio boils down to define a subvolume of indices. Here, the subvolumes we consider are obtained by selecting one smaller interval in one dimension of the NDRange. The *offset* is the first index of this interval of indices, the partitioning ratio defining the size of this interval. **Figure 2.15c** shows for a Sparse Matrix Vector Multiply (SpMV) the influence of the offset on performance when, for a sub-kernel with partitioning ratio 1/4, the offset is changed. The workload of this kernel is irregular. In the chosen sparse matrix, rows with a high index have more non-zero elements than those with a low index. This accounts for the execution time increase for large offsets, more than 7x the time of a 0-offset on CPU and 4x on GPU. When splitting a kernel into sub-kernels, this is a possible source of load-imbalance.

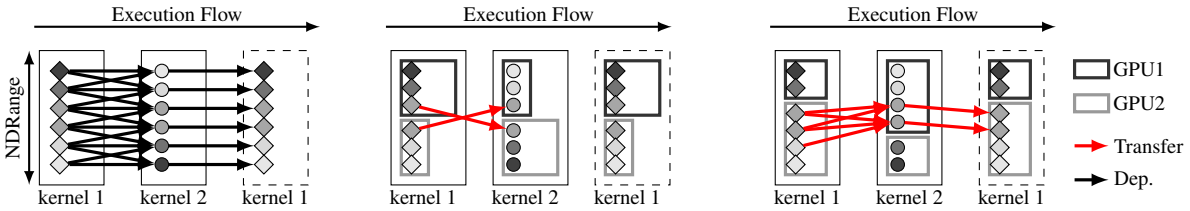
Impact of the iteration number

Many OpenCL kernels are executed in iterative computations. For instance, OTOO [48] is an astrophysics particle N-Body simulation and the same kernel is called repeatedly to compute forces and move the different particles. **Figure 2.15d** shows how the execution time changes for different iteration numbers and for different offsets, for each iteration of the computation. The kernel is split into 4 sub-kernels, each one is given a partitioning ratio of 1/4 and executed on one GPU. The input set corresponds to a non-uniform distribution of the masses in space. As this space is partitioned among the work-groups, this results in a non-homogeneous load distribution among the work-groups, changing with iteration number (dynamic workload).

Impact of communications

When partitioning a sequence of kernels to multiple devices, the partitioning of one kernel may have great impact on the amount of data to transfer between devices before executing another one when there is a data dependence between these kernels. We use an illustrative example to present some partitioning strategies and to show the impact of these strategies on data transfers.

Figure 2.16 illustrates two different strategies when partitioning a sequence of 2 kernels on 2 devices. The structure of the application is shown in **Figure 2.16a**. Threads from the iteration spaces of kernels 1 and 2 are respectively represented with diamonds and circles. The shades of gray represent the amount of work of each thread, dark threads have more work than light threads. Both kernels have irregular workload: for kernel 1 (resp. kernel 2), threads from the beginning of the iteration space have more work (resp. less work) than threads at the end of the iteration space. kernel 2 exhibits the structure of a stencil: each thread from its iteration space depends on the data produced by the thread from kernel 1 at the same position and also on the data produced by its two neighbor threads. For kernel 1 however, each thread only depends on the data produced by the thread at the same position in kernel 2.



(a) Application Structure. **(b)** Minimizing Transfers. **(c)** Minimizing Computation.

Figure 2.16: Impact on amount of data to transfer of different partitioning strategies.

Figure 2.16b illustrates a uniform partitioning strategy over 2 GPUs, where the two sub-kernels of kernel 1 and kernel 2 have a partitioning ratio of 0.5. A partitioning ratio of 0.5 on a device means that half of the iteration space of the kernel is executed on this device. For this specific application, this partitioning minimizes the amount of data to transfer. However, the execution times of the sub-kernels are imbalanced.

Figure 2.16c illustrates another partitioning that minimizes the computation time of each kernel. kernels 1 and 2 must then have different partitioning ratios to balance the execution time of

their sub-kernels. However, this partitioning implies much more data transfers, and may result in a slowdown.

This illustrative example shows the impact of the partitioning of each kernel from the sequence on the volume of data to transfer. Only considering the execution times of each kernel is not sufficient in order to minimize the overall execution time of a kernel sequence.

We now study the performance variation of two real applications (Jacobi and 2SpMV) for various kernel partitionings across two heterogeneous GPUs. Both applications consist in an iterated sequence of two data-dependent kernels. For each application, we measure the total execution time when the kernel sequence is iterated 100 times.

The Jacobi application consists in a first kernel corresponding to a 1D stencil kernel (here, k_1) followed by a memcpy from the output buffer to the input buffer (k_2). When partitioning this application on two devices, the volume of data to transfer between devices is minimized when both kernels have the same partitioning.

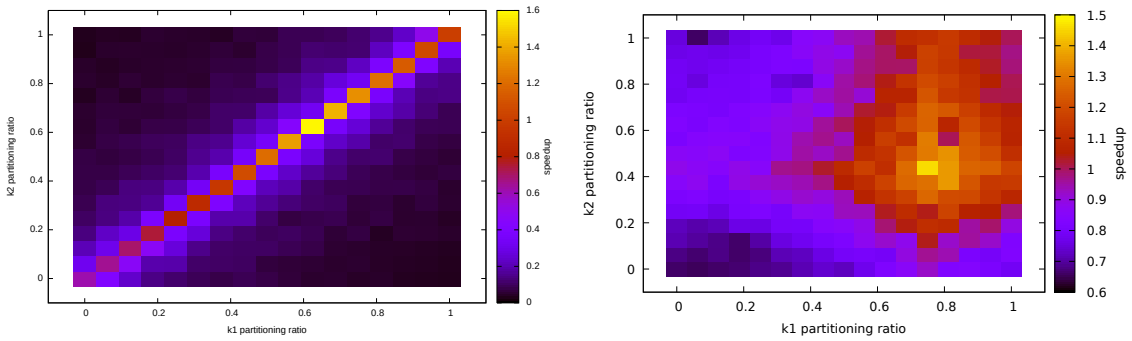
The 2SpMV application consists in a Sparse Matrix-Vector Multiplication applied on two different matrices, the output vector of one kernel is the input vector of the following one. Both kernels present irregular workload among threads, due to the sparsity structure of each matrix. In the matrix for k_1 (resp. k_2), rows with a low index have less (resp. more) non-0 elements than those with a high index. Each thread of a kernel only depends on data produced by the thread of the other kernel at the same position in the iteration space. When partitioning this application on two devices, the volume of data to transfer between devices is minimized when both kernels have the same partitioning.

Figures 2.17a (Jacobi) and **2.17b** (2SpMV) show the speedups according to the partitioning of the two kernels across two heterogeneous GPUs. The speedup shown is the speedup against the performance obtained by executing both kernels on the best device. The X-axis represents the partitioning ratio for k_1 on the first device, the Y-axis represents the partitioning ratio for k_2 on the first device. A partitioning ratio of 0.5 means that half of the iteration space of the kernel is executed on each device and a partitioning ratio of 1 means that the whole kernel is executed on the first device.

As we can see in **Figure 2.17a** if both kernels of the Jacobi application do not have the same partitioning ratio, the transfers between the kernels result in a significant slowdown (from 0.50 when the two kernels do not have the exact same partitioning ratio and up to 0.02 when the first kernel is entirely executed on the second device and the second kernel is entirely executed on the first device). Since the first device has better compute capabilities, the best speedup (1.6) is achieved by giving the partitioning ratio of 0.6 to both kernels.

For the 2SpMV application however, the best speedup (1.5) is achieved by giving a partitioning ratio of 0.7 to k_1 and a partitioning ratio of 0.4 to k_2.

These examples demonstrate that only minimizing the volume of transfers or only minimzing the sub-kernels execution time is not sufficient to minimize the overall execution time of a kernel sequence. The partitioning that minimizes the overall execution time is a trade-off between the balancing of the execution times of the sub-kernels of each kernel and the cost of the data transfers induced.



(a) Jacobi. **(b)** 2SpMV.

Figure 2.17: Partitioning a sequence of kernels.

2.2.3 Summary

This section presented the main challenges we need to address to automatically adapt a single-device application consisting in an iterated sequence of kernels to multiple devices.

First, we must automatically compute for each kernel, the read and written regions of the buffers passed as parameters depending on the slice of the NDRange executed onto each device. This would require a combination of static and dynamic analyses since some values required to compute these regions can be determined by a static analysis of the code whereas others are only available at runtime. Moreover, the analysis should be able to cope with codes containing indirect memory accesses. In addition, when partitioning a kernel whose NDRange has more than one dimension, the dimension split to distribute the parallel iteration space must be the one that allows to minimize the amount of data to transfer.

Second, the method should be able to cope with the heterogeneity of the hardware, but also with irregular workloads and dynamic load variations between repeated executions of the kernel sequence. In addition, the method has to take into account the communication times induced by the partitioning of each kernel in order to minimize the overall execution time of the application.

2.3 Principle of Adaptive Partitioning

This section presents the principles of our method to automatically adapt single-device OpenCL applications to heterogeneous multi-device architectures. The method we propose is threefold. First each kernel is analysed and a new version, *partition-ready*, is generated for each kernel at compile time. Then each time a kernel has to be executed, a partitioning is chosen based on previous executions if any, and the partition-ready kernel is instantiated on each device with the chosen partitioning. More precisely:

1. When a kernel code is first loaded, it is analysed and transformed into a partition-ready kernel which can execute only a slice of the original NDRange space. The objective of the analysis is to determine for each buffer accessed by the kernel the region it reads and writes depending on its NDRange. The analysis is performed once on the OpenCL code (no host code analysis) but the partition-ready kernel generated can be instantiated at runtime for any slice of the NDRange.

2. Each time a kernel is launched, a partitioning is determined. This partitioning determines the fraction of the original NDRange to execute onto each device. Then buffers regions are instantiated with the current partitioning and the actual parameters of the kernel. Depending on the result, all buffers are communicated to the devices or only the region they require. The same occurs for bringing back data from the devices.

3. The execution time of each kernel is collected for refining the partitioning in the possible following iterations.

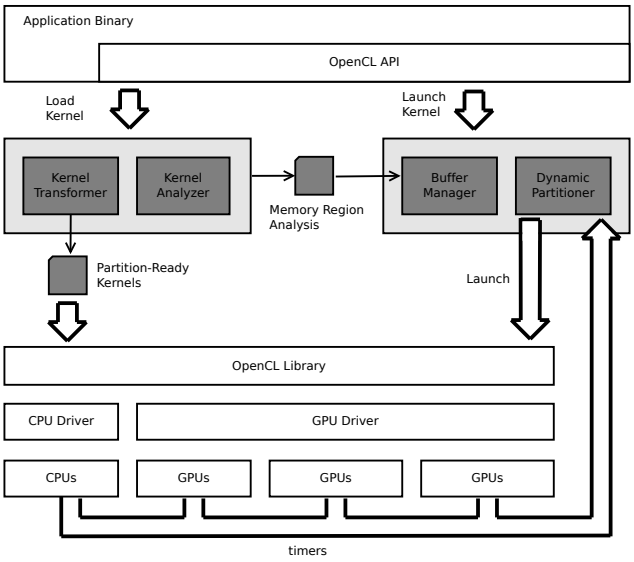


Figure 2.18: Framework Overview. At load time when a kernel is loaded, it is analysed by the *Kernel Analyzer* and a partition-ready kernel is generated by the *Kernel Transformer*. At runtime before a kernel is executed, the *Dynamic Partitioner* determine a partitioning for the kernel based on previous iteration if any and the *Buffer Manager* handle the necessary data transfers before sub-kernels execution.

The method we propose is completely transparent to the user and does not require any modification or recompilation of the original program. To this end our framework works by interposing calls to the OpenCL shared library from the application binary as shown in **Figure 2.18**. Our framework is composed of four components:

- The *Kernel Analyzer* is in charge of the static analysis of kernels.
- The *Kernel Transformer* performs source-to-source transformations of kernels.
- The *Dynamic Partitioner* determine the partitioning of each kernel.
- The *Buffer Manager* manages the necessary data transfers between devices.

We briefly present the static analysis and transformation performed by our analysis as well as the dynamic adaptation in **Sections 2.3.1 and 2.3.2**. Then, the management of buffers and data transfers is presented in **Section 2.3.3**. Finally, the general algorithm of the automatic transformation we want to achieve is presented in **Section 2.3.4**.

The memory region analysis to automatically partition kernels onto multiple devices is the subject of the **Chapter 3** and the dynamic load balancing is the subject of the **Chapter 4**.

2.3.1 Static Analysis and Transformation

At load time, when the sources of the program are loaded (`clCreateProgramWithSources`), the sources are compiled to LLVM bytecode in order to be analyzed by the *Kernel Analyzer*. The LLVM bytecode generated is only used by the *Kernel Analyzer*. Then the *Kernel Transformer* performs source-to-source transformations of the program. At compile time (`clBuildProgram`), these transformed sources will be compiled by each vendor instead of the original sources.

Memory Region Analysis

The *Kernel Analyzer* computes for each kernel parametric read and write regions for each buffer it accesses. These parametric regions correspond to the functions f_k and g_k presented in **Section 2.1.2**. These regions are obtained by an interval analysis and consist of a union of intervals defining the values of indices in the buffers that may be read/written by the kernel. These intervals may depend on thread ids, on the values of the scalar parameters of the kernels and on the values of other array elements in the case of indirections.

Partition-Ready Kernel Generation

The parallel iteration space of an OpenCL kernel is defined by a NDRange in 1, 2 or 3 dimensions. In order to distribute the computation of a kernel, its original NDRange is split along one of the dimensions as shown in **Figure 2.19**. Each device then executes the same kernel but only on a slice of the original NDRange.

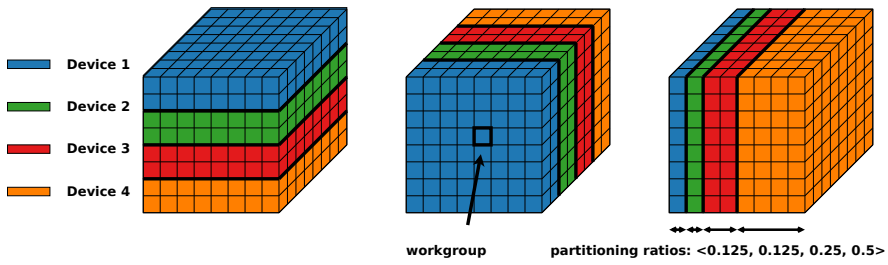


Figure 2.19: Splitting one dimension of a 3D NDRange to distribute the parallel iteration space onto 4 devices.

The OpenCL function `clEnqueueNDRangeKernel` to execute a kernel takes the four following parameters to define its NDRange:

- `work_dim`: the number of dimensions
- `global_work_offset`: the offset used to calculate the global id in each dimension
- `global_work_size`: the number of threads (or work-items) in each dimension
- `local_work_size`: the number of threads (or work-items) that make up a work-group

By reducing the `global_work_size` and setting the right `global_work_offset`, the OpenCL API allows to execute a kernel only on a slice of its original NDRange. However, when executing a kernel with a fraction of the original NDRange, some syntactic modifications are needed in order to keep the correct semantics. Indeed, the global size is different from the original kernel, the number of work-groups has changed and their ids have changed too. To keep the correct semantics, the *Kernel Transformer* performs a source-to-source transformation of each kernel. Two additional parameters are added to the kernels: `splitdim` (1, 2 or 3) accounts for the dimension of the NDRange that is split, and `numgroups` is the number of work-groups in this dimension. The details of this transformation are presented in **Section 4.4**.

2.3.2 Dynamic Adaptation

At runtime, the *Dynamic Partitioner* computes a new partitioning of the parallel iteration space defined by the NDRange of each kernel. Each device then executes the kernels on a fraction of their respective NDRange. These *sub-kernels* correspond to the partition-ready kernels instantiated with the *sub-NDRange* resulting from their partitioning. The goal of the *Dynamic Partitioner* is to determine at each iteration the partitioning of the whole sequence of kernels that minimizes the overall execution time of the sequence. The partitioning of the sequence at iteration $t + 1$ is determined using measures of kernels execution times at iteration t.

Each time a new partitioning of the kernel sequence is computed, the parametric read and write array regions associated to each sub-kernel are instantiated with the values of its scalar parameters, its current sub-NDRange and possibly the value of some array elements in case of indirections. The dimension of the NDRange that is split is selected at runtime depending on which one allows to distribute written data among devices, with no need for a merging operation, if possible.

The *Buffer Manager* then uses these instantiated regions to only transfer the data missing on each device.

2.3.3 Buffer Management

The *Buffer Manager* is responsible for the management of buffers at runtime. When the function `clCreateBuffer` is called, the *Buffer Manager* allocates a buffer of the requested size onto each device. These buffers are denoted as sub-buffers. The *Buffer Manager* then maintains a directory for each buffer in order to keep track, for each element of a buffer, of the list of devices that own this element in their own memory.

When the function `clEnqueueWriteBuffer` is called to transfer a memory region R from the host memory to a buffer B, the data transfer is delayed until kernels execution. Then, when a kernel reading buffer B is launched and the memory region analysis in instantiated for the chosen partitioning, only the required data is transferred from the host to each sub-buffer.

The *Buffer Manager* manages the required data transfers before the execution of each sub-kernel and only the data not already present on the devices are transferred.

After each kernel execution, the *Buffer Manager* updates the directory of each output buffer using the instantiated written region of each sub-kernel.

When the function `clEnqueueReadBuffer` is called the required data is transferred from one or several devices back to the host depending on which device owns the requested data.

Thus, the amount of data to transfer to each device is minimized. However, since we allocate a sub-buffer of the same size as the original buffer on each device, the memory space allocated on

each device is not reduced.

2.3.4 General Algorithm

The general algorithm of the dynamic adaptation of an iterative application with a sequence of m kernels onto n heterogeneous devices is shown in **Figure 2.20**.

First, at the beginning of each iteration, a new partitioning of the kernel sequence minimizing the total execution time of the iteration is computed based on the profiling of the previous iteration if any. This is the role of the statements lines 3 to 5.

Then the kernel sequence is executed and each device only executes a fraction of the original iteration space of each kernel.

Before executing the sub-kernels, only the minimum data transfers are performed. In order to avoid unnecessary data transfers, it is crucial to know for each device, which data are present in its own physical memory. To that end we define $directory_{\mathtt{B}}(x)$ as the function giving for each element x of a buffer `B`, the list of devices owning x. This function corresponds to the directory of buffer `B` maintained by the *Buffer Manager*.

The list of buffers read (resp. written) by a kernel k is denoted as $IN(kernel_k)$ (resp. $OUT(kernel_k)$) and the region of a buffer `B` read (resp. written) by threads of a sub-kernel of kernel k on device d is denoted as $f_k^{\mathtt{B}}(\mathcal{R}_k^d)$ (resp. $g_k^{\mathtt{B}}(\mathcal{R}_k^d)$).

Before executing a sub-kernel of kernel k on device d, for each buffer `B` read by this sub-kernel, the device d must have all elements of $f_k^{\mathtt{B}}(\mathcal{R}_k^d)$ in its own memory. Hence, the `parallel for` loop line 9 transfers in parallel for each device d, the missing data to execute its sub-kernel. The call to the function $transferBufferElement$ line 13 transfers the element at offset x in buffer `B` from a device owning x (given by $directory_{\mathtt{B}}(x)[0]$) to device d. Then the directory of `B` is updated to reflect its value after the data transfer.

After the required data have been transferred to each device, the parallel for loop line 19 executes each sub-kernel in parallel on the different devices. Then, the for loop line 23 updates the directory of all output buffers. After the execution of a sub-kernel of kernel k on device d, for each element x of a buffer `B` written by this sub-kernel, only the device d owns a valid version of x.

Finally, at the end of each iteration the sub-kernels execution times are collected as shown line 32.

```
1  for i = 0; i < iter; i = i + 1 do
2      /* compute a new partitioning of the kernel sequence based
          on previous iteration profiling                          */
3      for k = 0; k < m; k = k + 1 do
4          R_k^1 ... R_k^n ⟵ computePartition(R_k, timers_{i-1})
5      end
6      /* execute the kernel sequence                              */
7      for k = 0; k < m; k = k + 1 do
8          /* transfer the data missing on each device to execute
              the sub-kernels                                      */
9          parallel for d = 1; d ≤ n; d = d + 1 do
10             for B ∈ IN(kernel_k) do
11                 for x ∈ f_k^B(R_k^d) do
12                     if d ∉ directory_B(x) then
13                         transferBufferElement(B, x, directory_B(x)[0], d)
14                         directory_B(x) ⟵ directory_B(x) ∪ {d}
15                 end
16             end
17         end
18         /* execute the sub-kernels                              */
19         parallel for d = 0; d < n; d = d + 1 do
20             kernel_k(R_k^d)
21         end
22         /* update directories                                   */
23         for d = 1; d ≤ n; d = d + 1 do
24             for B ∈ OUT(kernel_k) do
25                 for x ∈ g_k^B(R_k^d) do
26                     directory_B(x) ⟵ d
27                 end
28             end
29         end
30     end
31     /* collect iteration profiling information                  */
32     timers_i = collectProfilingInfo()
33 end
```

Figure 2.20: General algorithm of the dynamic adaptation of a single-device iterative application with m kernels to n heterogeneous devices.

2.4 Summary

This chapter presents the challenges we need to address and the basic concepts of our method for automatic adaptation of single-device applications to heterogeneous architectures. The type of parallelism we are targeting consists in an iterative sequence of parallel loops. This allows the programmer to express the parallelism of his application with a sequence of parallel computation

kernels without worrying about the underlying architecture. The parallelization model we propose is to distribute the parallel iteration space over multiple devices. The two major challenges to automatically adapt a single-device application to heterogeneous multi-device architectures are the automatic data partitioning and the load balancing.

To address these challenges, the method we propose combines static and dynamic methods. At load time when the source code of the application is loaded, we perform a static analysis of the code to compute parametric regions of buffers accessed by kernels. Then the code of the application is transformed so that each kernel can execute only on a slice of the parallel iteration space. At runtime, a new partitioning of the kernel sequence is determined after each iteration, based on previous execution times and data transfers observed. Once the partitioning of a kernel has been determined, the parametric regions of the buffers accessed are instantiated with the current partitioning and other values determined at runtime in order to transfer to each device only the data required by the sub-kernel it executes and that are not already present in its own memory.

The next chapter presents our memory region analysis allowing to automatically partition the data accessed by complex kernels containing indirect memory accesses and atomic operations. Then the dynamic load balancing method we propose to minimize the overall execution time of an iterative application is presented in **Chapter 4**.

CHAPTER 3

Automatic Data Partitioning

4.1 Load Balancing Computation 76
4.1.1 Formalization 76
4.1.2 Resolution Method 77
4.1.3 Evaluation 78
4.2 Communication-Aware Load Balancing 81
4.2.1 Formalization 81
4.2.2 Evaluation 85
4.3 Related Work 88
4.3.1 Static Coarse-grained Partitioning 88
4.3.2 Dynamic Coarse-grain Partitioning 90
4.3.3 Fine-grained Partitioning 91
4.4 Implementation 91
4.5 Summary and conclusion of the first part 95

One of the main challenges when adapting a single-device application to multiple devices is to automatically partition the data across the devices so as to minimize the amount of data to transfer. The parallelization model we propose is to distribute the parallel iteration space of each kernel across the devices. For each kernel, each device only executes a sub-kernel of the original kernel. Hence, each device only requires the partial regions of input buffers read by its sub-kernel and only modifies the partial regions of output buffers written by its sub-kernel.

As explained in **Section 2.1.3**, when distributing kernels computation onto multiple devices, for input data it is possible to broadcast the whole buffers to every device. However, this solution implies unnecessary data transfers. Conversely, a precise analysis of the data required by each thread of the kernel allows to only transfer the required data to each device and results in shorter communication times. For output data, determining the regions written onto each device is even more important. Without a precise analysis of the region of buffers written by each device, we need to find a way to properly merge the partial results from different devices back to the host. One possible fallback solution is to transfer the whole output buffers from every device back to the host and then to execute a "merge" kernel in order to properly merge the modifications from each device. However, for many applications this solution may end up in a slowdown comparing to

executing the application on a single device, especially in the case of iterative applications, targeted in this part of this thesis.

This chapter presents our method to automatically partition the data of a single-device application to multiple devices. This method corresponds to the implementation of the *dsplit* function presented in **Figure 2.5b, page 30**. The method we propose is able to cope with complex kernels containing indirect memory accesses and atomic operations. We then demonstrate the effectiveness of our method on a molecular dynamics application called SOTL, containing complex kernels with indirect memory accesses and atomic operations. Finally, we compare our method to related works.

3.1 Memory Region Analysis

This section presents our memory analysis combining static and dynamic methods to determine the regions read/written by each sub-kernel when dealing with complex codes containing indirections and atomic operations.

3.1.1 Objectives and Principles

When splitting an OpenCL kernel into sub-kernels, the NDRange defining the parallel iteration space of the kernel is split into sub-NDRanges and each sub-kernel only executes thread belonging to its sub-NDRange.

The objective of our analysis is to precisely determine the region of each buffer read and written by each sub-kernel depending on the slice of the original parallel iteration space it executes.

Relying on the principles exposed in [49], our analysis consists in computing for each buffer parametric expressions of the memory locations read and written by each thread. Then, at runtime these parametric expressions are instantiated using an interval analysis to compute the memory regions accessed by a whole sub-kernel.

In this work we extend these parametric expressions to kernels with indirect memory accesses. When values inside an indirection array are increasing or decreasing, we propose the developer to annotate the indirection array with one of the two following pragmas:

- `#pragma opencl_increasing_buffer`
- `#pragma opencl_decreasing_buffer`

Thus, when a buffer B is indirectly accessed through an indirection array I and I is annotated with one of these pragmas, the region of buffer B accessed by a sub-kernel can be determined by only reading two values from the indirection array.

For example in the kernel code shown in **Figure 3.1**, the buffer `IA` is annotated as increasing. In this case the region of buffer `A` read by a sub-kernel whose sub-NDRange corresponds to the interval $[L, U]$ will be determined by our analysis by only reading the two values at indices L and $U + 1$ from buffer `IA`. The region will be approximated as the interval $[\texttt{IA}[L], \texttt{IA}[U + 1] - 1]$.

```
1  void SpatialBinning(float *A, int *IA, float *B, int n) {
2    int id = get_global_id(0);
3
4    float sum = 0;
5    int start = IA[id];
6    int end   = IA[id+1];
7
8    for (int i = start; i < end; i++)
9      sum += A[i];
10
11   B[id] = sum;
12
13   #pragma opencl_increasing_buffer IA
14 }
```

Figure 3.1: Indirection array annotation. In this example, the input buffer `A` is indirectly accessed through buffer `IA` and `IA` is annotated as increasing.

Our memory region analysis consists in two steps:

1. First, at load-time when kernel codes are compiled, a static analysis constructs parametric read and write regions for each buffer of each kernel. These expressions are parameterized by the ids of the NDRange, the scalar parameters of the kernel and by the values of input array elements.

2. Then at runtime, before a sub-kernel is executed these parametric regions are instantiated with the interval of thread ids it executes, the values of scalar parameters of the kernel and possibly the values of input array elements. Using an interval analysis, each parametric region is instantiated to a set of contiguous regions $[L_i, U_i]$.

The *Buffer Manager* uses these instantiated regions to only transfer the data required onto each device before sub-kernels execution. For each region $[L_i, U_i]$ of a buffer B read by a sub-kernel executed on device d, the *Buffer Manager* transfers this region to the sub-buffer of buffer B allocated on device d. Only elements of the region $[L_i, U_i]$ not already present on the device are transferred. Finally, for each buffer B, the *Buffer Manager* uses the instantiated write regions of each sub-kernel to keep track of the regions of B owned by each device.

As kernel threads are executed concurrently, each thread must write output buffers at different locations for the computation to be correct. Hence, when the regions of output buffers written by each sub-kernel are precisely determined by our analysis, no merging operation is required.

However, for kernels containing atomic operations, different threads can atomically modify the same location. This is problematic when two threads t_1 and t_2 belonging to different sub-kernels modify the same location. Indeed, since the sub-kernels are executed onto different devices, the atomic modifications carried out by thread t_1 is not visible from the device where t_2 is executed. Our method to address this issue is presented in **Section 3.1.4**.

When the code of a kernel is too complex and the region of a buffer accessed by each sub-kernel cannot be determined automatically by our analysis, the user can provide these regions through code annotations. The limits of our analysis and these annotations are presented in **Section 3.1.5**.

3.1.2 Parametric Region Construction and Instantiation

This section presents our method to build and instantiate the parametric regions of buffers accessed by each sub-kernel.

Parametric Region Construction

The parametric region construction is twofold: i) An inter-procedural alias analysis identifies all statements accessing buffers passed as parameter to the kernel. ii) For each load/store instruction from/to a buffer B a *Statement-Region* is computed and added to the *Parametric Read/Write Region* of B.

Definition 6. A *Statement-Region* is a guarded expression of the form $g_1(id), ..., g_n(id) : expr(id, s)$ with id a thread id, s the scalar parameters of the kernel, $g_i(id, s)$ a guard on a thread id (e.g.: $get\_global\_id(0) < expr$), and $expr(id, s)$ an *Index-Expression* on id and s.

Definition 7. An *Index-Expression* E is a typed expression with E either:

- a scalar parameter of the kernel,
- a thread id,
- a constant,
- an interval $[L, U]$ with L and U two expressions,
- $E_1\ op\ E_2$ with E_1, E_2 two expressions and $op \in \{+, -, *, /\}$,
- $[\![B, E, T]\!]$ the value of an element of type T from a buffer B at index E, with T a scalar type and E an index-expression,
- cast(T,E) a cast operation of expression E to type T,
- or *undef*.

Definition 8. A *Parametric Region* is a finite set of *Statement-Regions*.

Our analysis is based on the Static Single Assignment (SSA) form [45] where each variable is defined by exactly one statement in the program. Variables that are initially assigned in multiples statements are renamed into new instances, one per statement. When multiple control-flow paths join in the control-flow graph (CFG), renamed variables are combined with a ϕ-function into a new variable instance. This makes explicit use/def chains.

The *Statement-Region* of a statement S accessing a buffer is computed by following the use/def chain from S until reaching a call to a function returning a thread id, a scalar parameter of the kernel, a constant, an indirection or the result of an atomic operation.

When a ϕ-function is encountered in the use/def chain, if it is an induction variable and its interval of values can be determined, then the variable is replaced with its interval $[L, U]$. Otherwise, the variable is replaced with *undef*.

When the result of an atomic function is encountered in the use/def chain, the variable is replaced with *undef*.

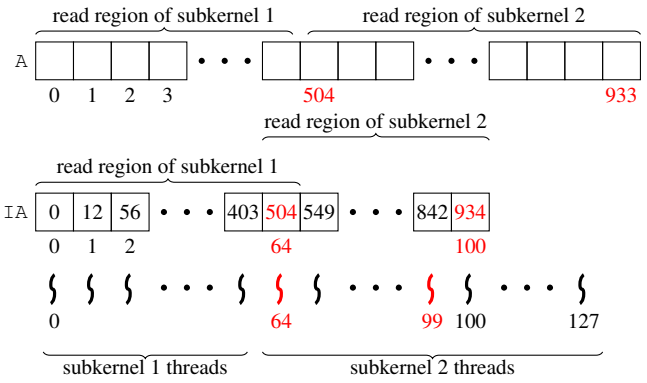
When a load instruction is encountered in the use/def chain (indirection) and the buffer B read by the load instruction has been annotated as increasing or decreasing, the variable is replaced with $[\![B, E, T]\!]$ where E is the expression of the index where B is read and T is the type of the element read. Otherwise, it is replaced with *undef*.

Indirections are not always integer values, hence to support all indirections we handle all basic OpenCL types (int, float, double, *etc*) and all cast operations.

The guards g_i of a *Statement-Region* correspond to the conditionals governing the execution of the corresponding load/store statement. These conditionals are computed using the *Iterated Post-Dominance Frontier* [45] of the load/store statement. Out of simplicity, only affine expressions on ids are kept, i.e. inequalities of the form: $a * id + b \leq c$ where a, b, c are expressions independent of thread ids with no indirection. All other conditionals are assumed to be true.

```
1  void SpatialBinning(float *A, int *IA, float *B, int n) {
2    int id = get_global_id(0);
3
4    if (id < n) {
5      float sum = 0;
6      int start = IA[id];
7      int end   = IA[id+1];
8
9      for (int i = start; i < end; i++)
10       sum += A[i];
11
12     B[id] = sum;
13   }
14
15   #pragma opencl_increasing_buffer IA
16 }
```

(a) code.



(b) Splitting buffer A.

Figure 3.2: SpatialBinning kernel Partitioning.

Example:

In order to construct the Parametric Read-Region of buffer `A` of the `SpatialBinning` kernel shown in **Figure 3.2a**, the analysis detects that there is only one load instruction reading buffer `A` corresponding to the statement line 10. Hence, the Parametric Read-Region of array

A contains only one *Statement-Region*. To compute this *Statement-Region*, the analysis detects that the load instruction read the element of array A at index i which is a ϕ-function function in the SSA form. This ϕ-function corresponds to an induction variable and is replaced with the interval [start, end − 1] corresponding to the bounds of the for loop line 9. The variable start is defined by a load instruction of an element of type int from buffer IA at index id where id is defined by get_global_id(0). Hence, this variable is replaced with $[\![\texttt{IA}, \mathit{global\_id}(0), \mathit{int}]\!]$. The same occurs with the variable end. Finally, the analysis computes the *Iterated Post-Dominance Frontier* of the statement line 10 and detects that this statement is only executed if the condition line 4 evaluates to true. The *Statement-Region* of buffer A built by the analysis is therefore:

$$\mathit{global\_id}(0) < \texttt{n} : \Big[[\![\texttt{IA}, \mathit{global\_id}(0), \mathit{int}]\!], [\![\texttt{IA}, \mathit{global\_id}(0) + 1, \mathit{int}]\!] - 1\Big].$$

Parametric Region Instantiation

The objective of the instantiation is to compute for each buffer B and each sub-kernel k the region of B accessed by all threads of k.

At runtime, the scalar parameters of the kernel and the *sub-NDRanges* of each *sub-kernel* are known, hence the *Parametric R/W Regions* can be instantiated. For each *Statement-Region*:

1. We inject the values of scalar parameters of the kernel along with its *sub-NDRange*.
2. The guards are applied by restricting the *sub-NDRange*.
3. If the expression E from an indirection $[\![B, E, T]\!]$ evaluates to an interval $[L, U]$ (i.e. E is not an indirection), $[\![B, E, T]\!]$ is replaced with $[val_1, val_2]$ with val_1 (resp. val_2) the value of the element of type T read from buffer B at index L (resp. U).

   This step is repeated until all indirections are replaced with their values.
4. A set of contiguous intervals $[L_i, U_i]$ is computed from the *Statement-Region* using interval arithmetic.

If a *Statement-Region* contains an *undef*, the whole buffer is considered: the *Statement-Region* is instantiated as $[0, N - 1]$ where N is the size of the buffer.

Example:

Figure 3.2b illustrates the partitioning of buffer A when the SpatialBinning kernel with original *NDRange* $[0, 127]$ is partitioned onto two devices with respective sub-NDRanges $[0, 63]$ and $[64, 127]$. Here the value of parameter n is 100.

In the following, we detail each step of the instantiation of the parametric read region of buffer A for sub-kernel 2:

1. At step 1, the value of n and the value of the sub-NDRange of the sub-kernel is injected into the *Statement-Region*. The *Statement-Region* becomes:

$$\mathit{global\_id}(0) < 100 : \Big[[\![\texttt{IA}, [64, 127], \mathit{int}]\!], [\![\texttt{IA}, [64, 127] + 1, \mathit{int}]\!] - 1\Big]$$

2. Then at step 2, the guard is evaluated:

$$\Big[[\![\texttt{IA}, [64, 99], int]\!], [\![\texttt{IA}, [64, 99] + 1, int]\!] - 1\Big]$$

and the expression is simplified:

$$\Big[[\![\texttt{IA}, [64, 99], int]\!], [\![\texttt{IA}, [65, 100], int]\!] - 1\Big]$$

As the values in buffer `IA` are increasing this expression evaluates as:

$$\Big[[v_1, v_2], [v_3, v_4] - 1]\Big]$$

with $v_1 = \texttt{IA}[64]$, $v_2 = \texttt{IA}[99]$, $v_3 = \texttt{IA}[65]$, $v_4 = \texttt{IA}[100]$ and is then simplified as:

$$\Big[v_1, v_4 - 1\Big]$$

3. At step 3 the values $v_1 = IA[64], v_4 = IA[100]$ are read from buffer `IA`. As shown in **Figure 3.2b**, the value of $IA[64]$ is 504 and the value of $IA[100]$ is 934.

4. In this example, the parametric read region of `A` only contains one *Statement-Region*, hence the final region computed at step 4 consists in the following single interval: $[504, 933]$.

3.1.3 Overlapping Write Regions

In certain case (e.g. complex guards not handled, over-approximation of the interval of an iteration variable), the analysis approximates the *write region* of a data buffer. When partitioning a kernel across multiple devices, this approximation may result in a region of the buffer potentially written by more than one sub-kernel.

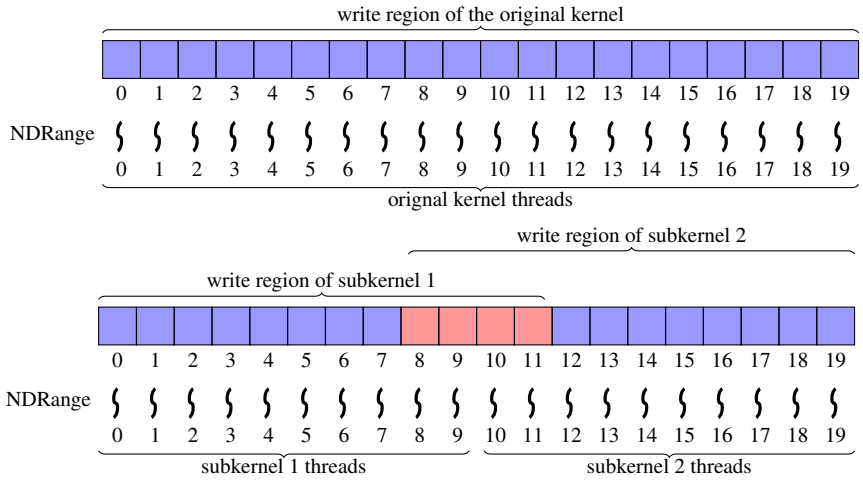
However, this situation is not possible. Since kernel threads are executed concurrently, each thread must write buffers at different locations for the kernel to be correct. Hence, the buffers regions actually written by each sub-kernel do not overlap.

When the *write regions* computed by our analysis for different sub-kernels are not disjoint, overlapping parts of the *write regions* are considered as *may-write region* whereas non-overlapping parts are considered as *must-write region*.

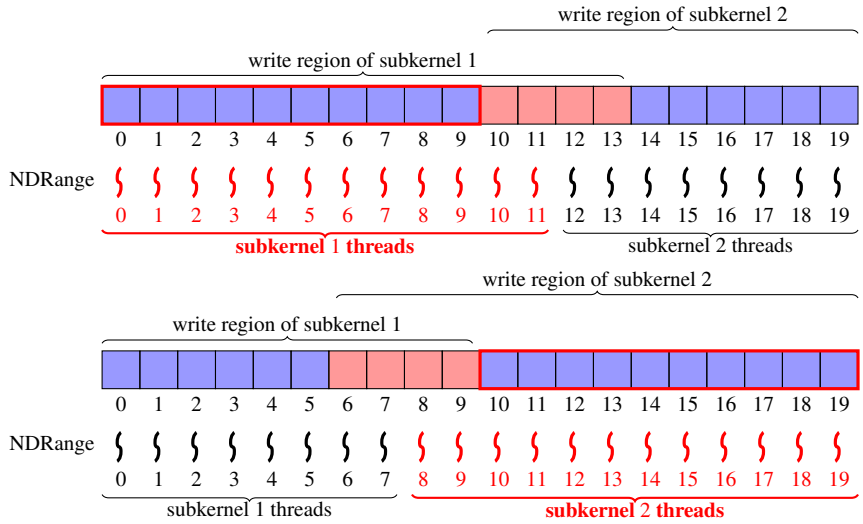
Figure 3.3a shows such an example. The sub-kernel on device 1 and the sub-kernel on device 2 (with respective sub-NDRanges $[0, 9]$ and $[10, 19]$) have non-disjoint write regions (respectively, $[0, 11]$ and $[8, 19]$) and so discontiguous *must-write regions*. The *must-write regions* is highlighted in blue in the figure whereas the *may-write region* is highlighted in red.

Our objective is to know which device owns the data in the *may-write region* in red ($[8, 11]$) after sub-kernels execution. In other words, we want to obtain *must-write regions* that cover the *must-write* region of the original kernel.

To tackle this issue, we propose to automatically increase the sub-NDRange of each sub-kernel and recompute its *must-write region* until contiguous *must-write regions* of the sub-kernels cover the *must-write regions* of the original kernel.



(a) Analysis Approximation Resulting in Non-Disjoint Write Regions.



(b) Analysis Results with Replication.

Figure 3.3: Write Region Analysis with Replication.

In the example shown in **Figure 3.3b**, we first increase the sub-NDRange of sub-kernel 1 and reduce the sub-NDRange of sub-kernel 2 and recompute their write-regions in order to increase the *must-write* region of sub-kernel 1. As shown in the figure, the write-region computed for sub-kernel 1 when its sub-NDRange corresponds to the interval $[0, 11]$ is $[0, 13]$, and the write-region computed for sub-kernel 2 when its sub-NDRange is $[12, 19]$ is $[10, 19]$. Hence, when the sub-NDRange of sub-kernel 1 is $[0, 11]$, we know that it writes at least the region $[0, 9]$ (*must-write* region) and may be also the region $[10, 13]$ (*may-write* region).

Then, we do the same for sub-kernel 2: We increase the sub-NDRange of sub-kernel 2 and reduce the sub-NDRange of sub-kernel 1 and recompute their write-regions in order to increase the *must-write* region of sub-kernel 2. As shown in the figure, the write-region computed for sub-kernel 2 when its sub-NDRange corresponds to the interval $[8, 19]$ is $[6, 19]$, and the write-region computed for sub-kernel 1 when its sub-NDRange is $[0, 7]$ is $[0, 9]$. Hence, when the sub-NDRange of sub-kernel 2 is $[10, 19]$, we know that it writes at least the region $[10, 19]$ (*must-write* region) and may be also the region $[6, 9]$ (*may-write* region).

When sub-kernel on device 1 and sub-kernel on device 2 have respective sub-NDRanges $[0, 11]$ and $[8, 19]$, their respective *must-write regions* ($[0, 9]$ and $[10, 19]$) cover the *must-write regions* of the original kernel ($[0, 19]$).

Hence, we overcome the possible approximations in the analysis by computation and data replication. In **Figure 3.3b**, the sub-NDRange $[8, 11]$ is executed by both devices. Threads from sub-kernel 1 may potentially write the region $[10, 13]$ and threads from sub-kernel 2 may potentially write the region $[6, 9]$. However, the *Buffer Manager* considers that after sub-kernels execution only device 1 owns elements from region $[0, 9]$ and only device 2 owns elements from the region $[10, 19]$.

3.1.4 Atomics

To handle atomic operations when distributing a kernel over multiple devices, two cases must be considered. When the result of an atomic operation is used inside the kernel, to ensure the correctness of the computation it is necessary that all threads modifying the same region of the buffer execute on the same device. To tackle this issue the atomic instruction is considered as a store instruction. This results in regions written by possibly multiple devices (*may-write regions*) as in **Figure 3.3a** and is solved by replication, as in **Figure 3.3b**.

When the result of the atomic operation is not used inside the kernel, the analysis checks whether the atomic operation is commutative or not. When the atomic operation is not commutative, it is considered as a store instruction as previously. Instead, when the atomic operation is commutative, the kernel analysis is augmented with *atomic-regions*. The region of a buffer modified by the atomic operation is tagged as an *atomic-region* instead of being a *write-region*. If the *atomic-region* of different sub-kernels overlap, the partial results from the overlapping regions on all devices are read back to the host after sub-kernels execution. Then a reduction is performed to merge the results properly.

3.1.5 Limits

In the following situations, our memory region analysis does not allow capturing precisely the read and write regions of a kernel:

- Indirection: When a buffer is indirectly accessed through an indirection array I and values in I are not increasing or decreasing, our analysis is not capable of determining the region accessed by each sub-kernel.

- Complex control-flow: When the index where a load/store instruction accesses a buffer corresponds to a ϕ-instruction in the SSA-form and this ϕ-instruction does not correspond to an induction variable, our analysis is not capable of building a parametric expression. In addition, the bounds of iteration variable cannot always be computed.

- Multi-dimensional regions: As our memory region analysis resort to an interval analysis, when the region of a buffer accessed by a kernel is indexed in more than one dimension, the region is over-approximated.

- Complex guard: When the conditional governing the execution of a store/load instruction into/from a buffer is too complex, it is ignored by our analysis.

As a fallback solution, the user can provide the region of a buffer `B` read, written or atomically modified through the following pragmas:

- `pragma read B[expr1,expr2]`
- `pragma write B[expr1,expr2]`
- `pragma atomic B[expr1,expr2]`

These annotations are contextual and the bounds of the regions provided through these annotations (`expr1` and `expr2`) can refer to other variables in the code. The user can also add additional code in the kernel that will only be used for the annotations. This additional code will be removed by dead code elimination when the kernel will be compiled and will therefore have no impact on the code actually executed on each device.

3.2 Case Study: SOTL

In this section, we detail how our method is able to automatically partition a real application with indirect memory accesses and atomic operations called SOTL.

SOTL is an OpenCL simulation of interactions between particles based on the Lennard-Jones potential [50], widely used in the field of molecular physics. This potential energy is often used to capture both attraction phenomena between atoms when they are distant, and repulsion phenomena when they are too close. An overview of the rendering obtained with this application when simulating the collision of two projectiles with a material is shown in **Figure 3.4**.

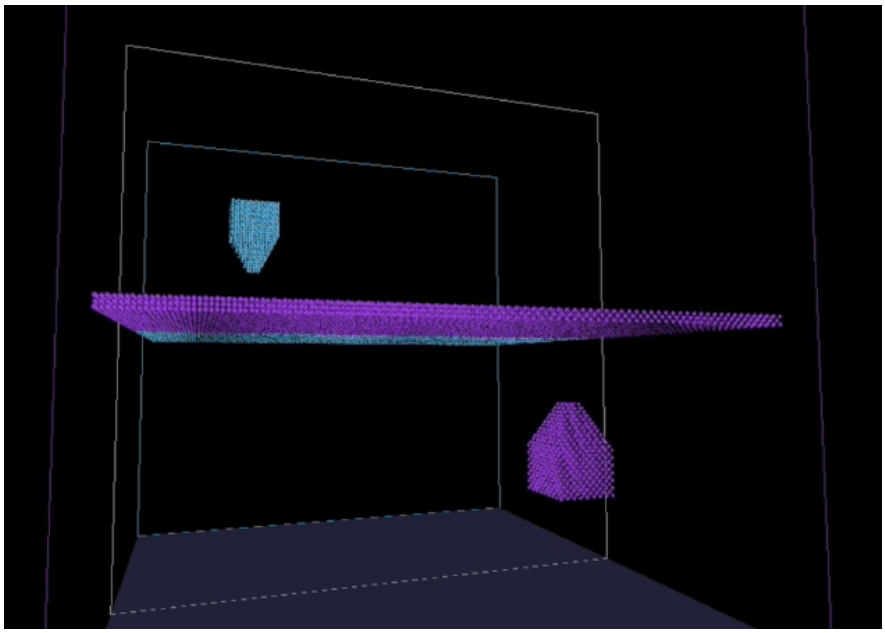


Figure 3.4: Overview of SOTL.

In this simulation, forces are only calculated between particles that are within a small region corresponding to the *cut-off radius* (denoted r_c). To calculate the result of the interactions between atoms, the application memorizes for each atom its position (x, y, z) and its speed (dx, dy, dz). To

simplify, the velocity of an atom is calibrated by a time step $\Delta_t = 1$. Hence, at each iteration of the simulation, (dx, dy, dz) represents the vector that must be added to the position of an atom to obtain its position at next iteration. The intensity F_{ij} of a force applied by an atom j to an atom i is given by the following formula:

$$F_{ij} = \begin{cases} 24\frac{\epsilon}{r}\left(\left(\frac{\sigma}{r}\right)^6 - \left(\frac{\sigma}{r}\right)^{12}\right) & \text{if } r \leq r_c \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

where r is the distance between i and j. σ and ϵ are constants chosen to respect the physical properties of the simulated material.

When the distance between a pair of atoms is greater than the cut-off radius r_c, the forces are neglected. Thus, at each iteration, it is necessary to know the particles that are within this cut-off radius in order to avoid unnecessary calculations.

The technique used in SOTL to reduce the number of computation is based on a subdivision of the 3D space into cubes of size d called *cells*, where d is equal to the cut-off radius r_c as shown in **Figure 3.5**. Thus, the neighbors of an atom are necessarily found in the 26 cells surrounding the cell containing the atom (plus its cell itself). This gives a total of 27 cells to go through to compute the forces applied to an atom.

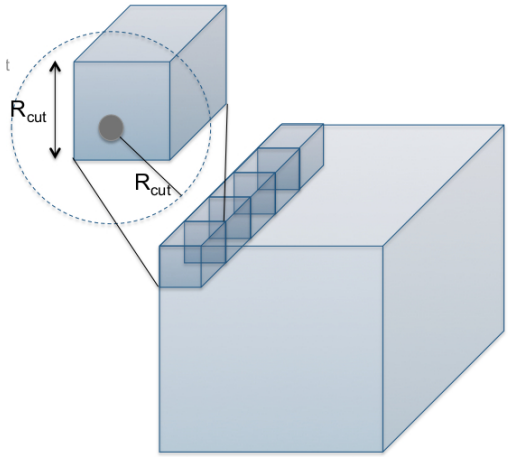


Figure 3.5: Subdivision of the 3D space of the domain into cells.

3.2.1 Algorithm and Data Structures

During the simulation phase, particles interact and move through the domain. At each iteration, it is assumed that an atom cannot move further than its neighboring cells. Thus, using an appropriate data structure, the search for neighboring particles is limited to the 27 related cells. However, this approach requires sorting all atoms at the beginning of each iteration before computing the forces.

This technique to accelerate the neighboring particle search is called spatial binning [46] and is widely used in particle simulation applications.

The general algorithm of this simulation of interactions between atoms using the Lennard Jones' potential consists in sorting the atoms and then computing the forces and moving the particles at each iteration.

Data Structures

Each cell from the domain has an identifier in each dimension ($cell\_id_x, cell\_id_y, cell\_id_z$), and a global identifier *cell_id* computed as follows:

$$cell\_id = cell\_id_x * cells_x * cells_y + cell\_id_y * cells_x + cell\_id_y \tag{3.2}$$

where $cells_i$ is the total number of cells in dimension i.

The cell id of an atom in all axes is computed from its position as follows:

$$\begin{aligned} cell\_id_x &= pos_x / r_c \\ cell\_id_y &= pos_y / r_c \\ cell\_id_z &= pos_z / r_c \end{aligned} \tag{3.3}$$

where pos_i is the coordinate of the atom in dimension i, and r_c the cut-off radius.

To facilitate the search for the nearest neighbors, the positions of the atoms are sorted according to the cell to which they belong and an indirection array `cells` allow accessing atoms from a given cell.

When atoms are sorted, the index of the first atom of the cell with global identifier i in the buffer containing the positions is given by `cells[i]`. Hence, the number of atoms contained in this cell is given by `cells[i+1]-cells[i]` and all atoms belonging to this cell are stored contiguously from the index `cells[i]` in the buffer of positions. The size of buffer `cells` is $nc + 1$ where nc is the number of cells in the domain of the simulation.

Figure 3.6 illustrates this data structure. In this example the cell with global identifier 2 contains 3 atoms and the index of the first atom of this cell in the position array is $cells[2] = 4$.

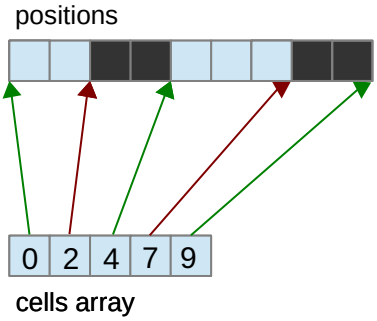


Figure 3.6: SOTL: Cells array.

The coordinates and speeds of the atoms are stored in two different buffers: `pos` and `speed`. The coordinate of the i-th atom is given by `pos[i]` and its speed by `speed[i]`. Two other buffers, `sorted_pos` and `sorted_speed`, are used to sort the atoms.

At the beginning of each iteration, positions and speeds of atoms computed at previous iteration are stored into buffers `pos` and `speed`. As each atom may have moved to one of its neighboring cells after each iteration, the cells array is rebuilt at the beginning of each iteration. Then, the positions and speeds of atoms computed at previous iteration are sorted according to the cell to which they belong into buffers `sorted_speed` and `sorted_pos`.

Algorithm

The algorithm of SOTL is based on the successive execution of the following OpenCL kernels:

- *reset*: reset the cell array
- *count*: count the number of atoms per cell
- *scan*: compute the index of the first atom of each cell using a prefix sum
- *copy*: backup cell array
- *sort*: sort the atoms
- *force*: compute forces and update atoms positions

The data dependences between these kernels is shown in **Figure 2.4, page 29**.

At the beginning of iteration t, buffers `pos` and `speed` contain the positions and speeds of the atoms computed at iteration $t-1$. The atoms are not sorted since at each iteration, each atom can move to another cell.

The first step of the algorithm consists in computing the number of atoms in each cell. This is the role of the two first kernels *reset* and *count*. The *reset* kernel initializes array `cells` to zero. Then the *count* kernel computes for each atom the global identifier *cell_id* of the cell to which it belongs from its position and increment the element of array `cells` at index *cell_id*. At the end of this step, the value of `cells[`i`]` is the number of atoms in the cell with global identifier i.

The second step of the algorithm consists in calculating the index where the first atom of each cell must be stored in array `sorted_pos` by computing the prefix sum of array `cells`. This the role of the *scan*. At the end of this step, the interval [`cells`$[i]$, `cells`$[i+1]-1$] correspond to the indices where atoms belonging to the cell with global identifier i must be stored in the `sorted_pos` array.

The third step consists in sorting the atoms. This is the role of the two next kernels. The *copy* kernel makes a backup of the `cells` array which will be modified by the *sort* kernel. Then the *sort* kernel reads for each atom its position and speed from buffers `pos` and `speed` and copy them at the index corresponding to the global identifier of the cell to which the atom belong in buffers `sorted_pos` and `sorted_speed` as illustrated in **Figure 3.7**.

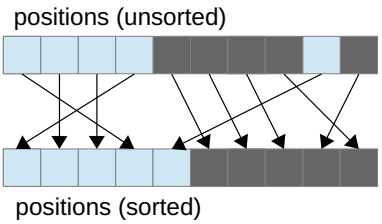


Figure 3.7: SOTL: Sorting atoms.

The last step consists in computing the forces applied to each atom to update its speed and position. This is the role of the *force* kernel. At each iteration, atoms cannot move further than their neighboring cells. For each atom, this kernel only calculates interactions with atoms belonging to its neighboring cells in order to avoid unnecessary computation.

3.2.2 Kernels Analysis

We detail here how our memory region analysis automatically partition the data for the three most complex kernels of SOTL: *count*, *sort* and *force*. The three other kernels do not contain indirect memory access nor atomic operation and do not require the techniques presented in **Sections 3.1.3, 3.1.4 and 3.1.5** to be partitioned. For sake of clarity, we present our memory region analysis on an 1D version of SOTL where the position and speed arrays only contains the position and speed of atoms in the x-axis and the cut-off radius r_c is 1. Hence, the position of an atom correspond to its cell id. Nevertheless, our analysis also works with the 3D version of SOTL as shown in the next section.

Partitioning the *force* kernel

The code of the *force* kernel from SOTL is shown in **Figure 3.8**. The size of the NDRange corresponds to the number of particles in the domain and each thread computes in parallel a new position and a new speed for the particle corresponding to its index in the NDRange.

```
1  #define CELL_SIZE 1
2
3  kernel force(float *sorted_pos,   float *pos,
4               float *sorted_speed, float *speed, int *cells) {
5    int gid = get_global_id(0);
6    float current_pos = sorted_pos[gid];
7
8    // Get cell id of current atom
9    int cell_id = current_pos  / CELL_SIZE;
10
11   float force_total = 0;
12
13   // Iterate through neigbor cells to compute the forces applied on the
14   // current atom
15   for (int i=cells[cell_id-1]; i<cells[cell_id+2]; i++) {
16     float dist2 = (sorted_pos[i] - sorted_pos[gid]) *
17                   (sorted_pos[i] - sorted_pos[gid]);
18     if (dist2 < LENNARD_SQUARED_CUTOFF)
19       force_total +=  ...;
20   }
21
22   // Update speed
23   speed[gid] = sorted_speed[gid] + force_total;
24
25   // Update position
26   pos[gid] = current_pos + speed[gid] * DELTA_T;
27
28   // Safe indirections
29   #pragma opencl_increasing_buffer sorted_pos, cells
30 }
```

Figure 3.8: *force* kernel from SOTL.

More precisely, each thread reads the position of the atom corresponding to its index in the NDRange from buffer `sorted_pos` as shown in line 6 in the code. The cell id of the atom is

calculated line 9 by dividing its position by the cell size. Then, the force applied to the atom is calculated in the for loop line 15 by iterating over the positions of atoms contained in the neighboring cells. Finally, the updated speed and position are stored into buffers `speed` and `pos` lines 23 and 36.

In this kernel, buffer `cells` is indirectly accessed through buffer `sorted_pos` line 15 and buffer `sorted_pos` is indirectly accessed through buffer `cells` lines 16 and 17.

Nevertheless, at this step of the algorithm, the atom positions contained in buffer `sorted_pos` have been sorted according to the cell to which they belong. Hence, values in buffers `cells` and `sorted_pos` are increasing. The developer can therefore annotate these buffers as increasing as shown in line 29.

For this kernel, splitting buffers `pos`, `speed` and `sorted_speed` is straightforward since each thread only accesses these buffers at the location corresponding to its index in the NDRange (`get_global_id(0)`). Splitting buffers `cells` and `sorted_pos` is more complicated as they are accessed through indirection arrays. **Figure 3.9** illustrates the partitioning of these buffers computed by our analysis when this kernel is partitioned onto 3 devices whose sub-NDRanges corresponds the intervals $[0, S_1 - 1]$, $[S_1, S_2 - 1]$ and $[S_2, S_3 - 1]$ respectively.

In order to partition these arrays, our static analysis automatically determines that the region of buffer `sorted_pos` read by each thread corresponds to the parametric region r_1, and the region of buffer `cells` read by each thread corresponds to the parametric region r_2.

With:

- r_3: the index where buffer `sorted_pos` is read at line 6
- r_4: the interval of values taken by the induction variable `i` in the for loop line 15
- r_5 and r_6: the values returned by the two load statements reading buffer `cells` line 15.
- r_7 and r_8: the values of `cell_id-1` and `cell_id+2`.

The regions r_1 and r_2 are defined as follows:

- $r_1 = r_3 \cup r_4$
- $r_2 = r_7 \cup r_8$
- r_3 = `get_global_id(0)`
- $r_4 = [r_5, r_6 - 1]$
- r_5 = `cells`$[r_7]$
- r_6 = `cells`$[r_8]$
- r_7 = `(int) sorted_pos[get_global_id(0)]/1-1`
- r_8 = `(int) sorted_pos[get_global_id(0)]/1+2`

Let's know explain how these regions are instantiated at runtime for sub-kernel 2. First the interval defining its sub-NDRange is injected into these regions:

- $r_1 = r_3 \cup r_4$

- $r_2 = r_7 \cup r_8$
- $r_3 = [\mathbf{S_1}, \mathbf{S_2} - \mathbf{1}]$
- $r_4 = [r_5, r_6 - 1]$
- $r_5 = \texttt{cells}[r_7]$
- $r_6 = \texttt{cells}[r_8]$
- $r_7 = \texttt{(int) sorted\_pos}[[\mathbf{S_1}, \mathbf{S_2} - \mathbf{1}]]/1 - 1$
- $r_8 = \texttt{(int) sorted\_pos}[[\mathbf{S_1}, \mathbf{S_2} - \mathbf{1}]]/1 + 2.$

Then, as buffer `sorted_pos` is annotated as increasing, regions r_7 and r_8 can be calculated by only considering the values of buffer `sorted_pos` at indices S_1 and $S_2 - 1$:

- $r_7 = [\texttt{sorted\_pos}[S_1], \texttt{sorted\_pos}[S_2 - 1]] - 1$
- $r_8 = [\texttt{sorted\_pos}[S_1], \texttt{sorted\_pos}[S_2 - 1]] + 2$

Hence, at runtime only the two elements at indices S_1 and $S_2 - 1$ are read from buffer `sorted_pos` to instantiate these regions:

- $r_7 = [9, 20] - 1 = [8, 19]$
- $r_8 = [9, 20] + 2 = [11, 22]$

Then, instantiated regions r_7 and r_8 are injected into parametric regions r_5 and r_6:

- $r_5 = \texttt{cells}[[8, 19]]$
- $r_6 = \texttt{cells}[[11, 22]]$

As buffer `cells` is annotated as increasing, region r_4 can be calculated by only considering the values of buffer `cells` at indices 8 and 22:

- $r_4 = [\texttt{cells}[8, 19], \texttt{cells}[11, 22] - 1] = [\texttt{cells}[8], \texttt{cells}[22] - 1]$.

Hence, at runtime only the two elements at indices 8 and 22 are read from buffer `cells` whose values are $S_1 - 3$ and $S_2 + 2$ respectively. Therefore, the region is instantiated as $r_4 = [S_1 - 3, S_2 + 1]$. Finally, we obtain the following regions for r_1 and r_2:

- $r_1 = [S_1, S_2 - 1] \cup [S_1 - 3, S_2 + 1] = [S_1 - 3, S_2 + 1]$
- $r_2 = [8, 19] \cup [11, 22] = [8, 22]$

As we can see in **Figure 3.9**, the regions of buffers `cells` and `sorted_pos` read by each sub-kernel are not disjoint, hence elements highlighted in blue are replicated on multiple devices. In contrast, the regions of buffer `pos` written by each sub-kernel are disjoint. Hence, the analysis knows that after sub-kernels execution, the regions $[0, S - 1]$, $[S_1, S_2 - 1]$ and $[S_2, S_3 - 1]$ of buffer `pos` are only owned by devices 1, 2 and 3 respectively.

After sub-kernel executions, each atom may have moved to one of its two neighboring cells. Hence, positions in buffer `pos` are not sorted. **Figure 3.9** illustrates possible values in buffer `pos` after sub-kernels execution. As the cell size is 1 in this example, values in buffer `sorted_pos` and `pos` correspond to the cell id of the atoms.

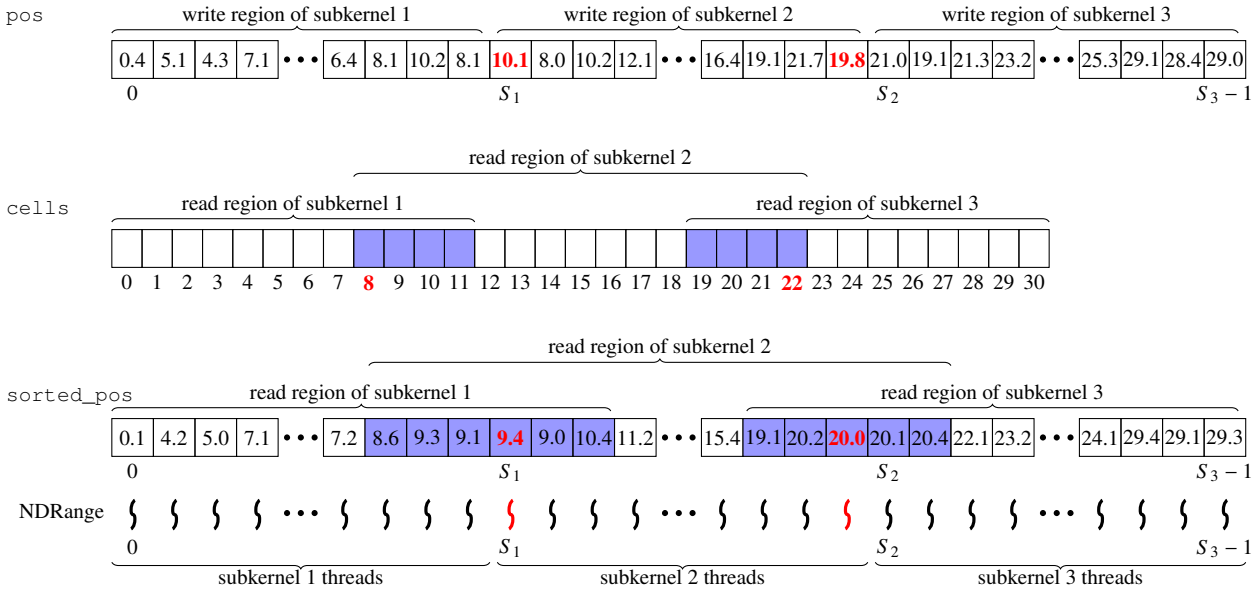


Figure 3.9: Partitioning arrays from the *force* kernel.

Partitioning the *count* kernel

The *count* kernel from SOTL is shown in **Figure 3.10**. The role of this kernel is to count the number of atoms in each cell. This kernel takes as parameter two arrays: `pos` containing the position of each atom and `cells` the output array initialized to zero in which the number of atoms per cell will be calculated.

The NDRange of this kernel corresponds to the number of atoms in the domain. Each thread reads the position of the atom whose index correspond to its index in the NDRange and computes its cell id as shown line 5. Then the element of the `cells` array whose index is the cell id of the atom is incremented atomically as shown line 8.

```
1  kernel count(float *pos, int *cells, int N) {
2    int gid = get_global_id(0);
3
4    // Get cell index of current atom
5    int cell_id = pos[gid] / CELL_SIZE;
6
7    // Increment the number of atom in the cell
8    atomic_inc(&cells[cell_id]);
9
10   // User Hint
11   #pragma atomic cells [cell_id-2:cell_id+2]
12 }
```

Figure 3.10: *count* kernel from SOTL.

For this kernel splitting the `pos` array is straightforward since each thread only reads this array at the index corresponding to its index in the NDRange. However splitting `cells` array is more complicated.

At this step of the algorithm, values in buffer `pos` are not sorted since each atom may have moved to one of its two neighboring cells at previous iteration. Hence, the region of buffer `cells` touched by a sub-kernel of sub-NDRange $[L, U]$ does not match the interval $[\text{cell\_id}_L, \text{cell\_id}_U]$ with cell_id_L and cell_id_U the cell ids of atoms stored at indices L and U in buffer `pos`. This is illustrated in **Figure 3.11** where $\text{cell\_id}_{S_1} = 10$ and $\text{cell\_id}_{S_2-1} = 19$, while the region of buffer `cells` touched by sub-kernel 2 corresponds to the interval $[8, 21]$.

However, given two threads i and j with $i \leq j$, it is possible to compute a lower bound for cell_id_j based on cell_id_i and similarly an upper bound for cell_id_i based on cell_id_j. Hence it is possible to calculate an over-approximation of the region of buffer `cells` touched by a sub-kernel of sub-NDRange $[L, U]$ by only considering the values of buffer `pos` at indices L and U.

Demonstration: At iteration $t-1$, before executing the *force* kernel, atoms are sorted according to the cell to which they belong. Hence, when $i \leq j$, then $\text{cell\_id}_i^{t-1} \leq \text{cell\_id}_j^{t-1}$. With cell_id_i^{t-1} and cell_id_j^{t-1} the cell ids of atoms stored at indices i and j in buffer `sorted_pos` at iteration $t-1$. Then, after execution of the *force* kernel each atom may have moved to another cell. However, as atoms cannot move further than their neighboring cells at each iteration, we obtain the following inequalities:

$$\begin{cases} \text{cell\_id}_i^{t-1} \leq \text{cell\_id}_j^{t-1} \\ \text{cell\_id}_i^{t-1} - 1 \leq \text{cell\_id}_i^{t} \leq \text{cell\_id}_i^{t-1} + 1 \\ \text{cell\_id}_j^{t-1} - 1 \leq \text{cell\_id}_j^{t} \leq \text{cell\_id}_j^{t-1} + 1 \end{cases} \implies \begin{cases} \text{cell\_id}_i^{t} - 2 \leq \text{cell\_id}_j^{t} \\ \text{cell\_id}_i^{t} \leq \text{cell\_id}_j^{t} + 2 \end{cases}$$

with cell_id_i^t and cell_id_j^t the cell ids of atoms stored at indices i and j in buffer `pos` at iteration t. Therefore, the region of buffer `cells` touched by a sub-kernel of sub-NDRange $[L, U]$ can be approximated with the interval $[\text{cell\_id}_L - 2, \text{cell\_id}_U + 2]$.

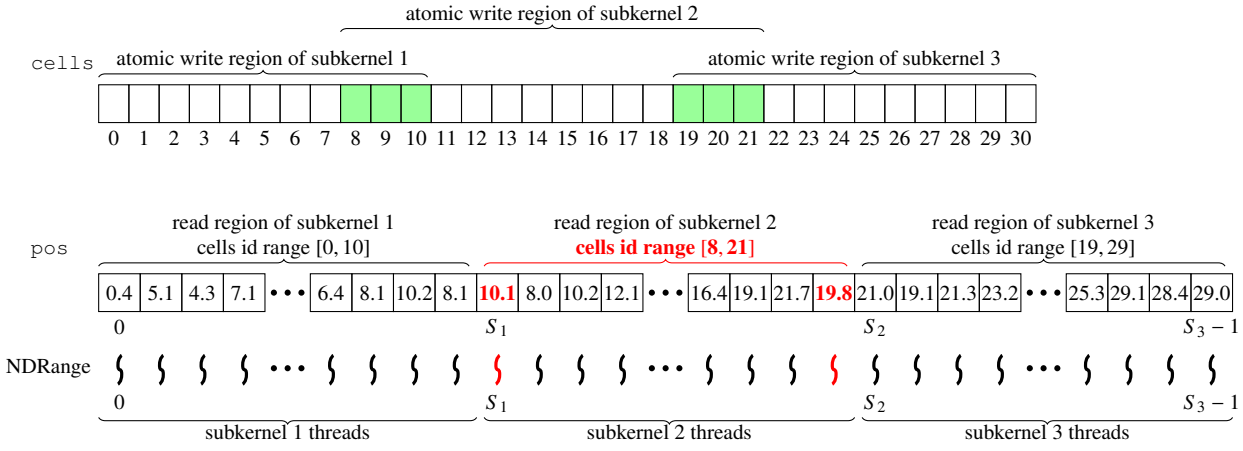


Figure 3.11: Partitioning arrays from the *count* kernel.

This physical property cannot be determined automatically by our analysis, but the developer of this application can annotate the kernel as shown line 11 in the code. In this case, only the two elements at indices L and U will be read from buffer `pos` at runtime in order to compute the region of buffer `cells` touched by a sub-kernel of sub-NDRange $[L, U]$.

As highlighted in green in **Figure 3.11**, the analysis of this kernel may result in regions of buffer `cells` touched by multiple devices. Indeed, threads in charge of atoms belonging to the same cell can execute onto different devices. In that case the atomic operation is commutative and the result of the atomic region is not used inside the kernel. Therefore, after sub-kernels execution only the partial results from the overlapping region in green are read back from all devices to the host. Then a reduction is performed to merge these partial results properly as explained in **Section 3.1.4**.

Partitioning the *sort* kernel

The *sort* kernel from SOTL is shown in **Figure 3.12**. The role of this kernel is to read the position and speed of each atom from the unsorted buffers `pos` and `speed` and copy them at the right location into sorted buffers `sorted_pos` and `sorted_speed`.

The NDRange of this kernel corresponds to the number of atoms in the domain. It takes five buffers as parameters: `pos`, `speed`, `sorted_pos`, `sorted_speed` and `cells`.

At the end of this kernel, the positions and speeds of atoms belonging the cell of id *i* must be stored contiguously from index `cells`[*i*] into buffers `sorted_pos` and `sorted_speed`.

To that end, each thread first reads the position of the atom corresponding to its index in the NDRange and computes its cell id as shown line 6. Then, the location where the position and speed of the atom must be stored in the sorted buffers is given by `cells[cell_id]`. To ensure that all atoms belonging to the same cell are not copied at the same location, the value of `cells[cell_id]` is incremented atomically by each thread as shown in line 9.

The atomic instruction line 9 first reads the value of `cells[cell_id]` (referred to as *old*) then stores value *old* + 1 in `cells[cell_id]` and returns *old*.

Finally, the position and speed of the atom are copied at the right location into buffers `sorted_pos` and `sorted_speed` as shown lines 12 and 13.

```
1  kernel sort(float *pos,   float *sorted_pos,
2              float *speed, float *sorted_speed, int *cells) {
3    int gid = get_global_id(0);
4
5    // Get cell index of current atom
6    int cell_id = pos[gid] / CELL_SIZE;
7
8    // Get index of the next free case in the cell
9    int shift = atomic_inc(&cells[cell_id]);
10
11   // Copy atom position and speed in the right cell of the
         sorted buffers
12   sorted_pos[shift] = pos[gid];
13   sorted_speed[shift] = speed[gid];
14
15   // User Hints
16   int start_id = cells[cell_id-2], end_id = cells[cell_id+3] -1;
17   #pragma write sorted_pos [start_id:end_id]
18   #pragma write sorted_speed [start_id:end_id]
19   #pragma atomic cells[cell_id-2:cell_id+2]
20 }
```

Figure 3.12: *sort* kernel from SOTL.

For this kernel splitting the `pos` and `speed` arrays is straightforward since each thread only reads these arrays at the location corresponding to its index in the NDRange. However, determining the regions of other buffers accessed by each sub-kernel is impossible without any hint from the user. Indeed, the location where each thread writes into buffers `sorted_pos` and `sorted_speed`, (`shift` line 9 in the code), is the result of an atomic operation on an unsorted buffer. Hence, determining its value is impossible without executing the kernel.

Nevertheless, as explained previously, the developer of the application knows that given a region $[L, U]$ of buffer `pos`, the cell ids of atoms stored in this region can be over-approximated with the interval $[\text{cell\_id}_L - 2, \text{cell\_id}_U + 2]$. Hence, the region where the positions and speeds of these atoms will be stored in buffers `sorted_pos` and `sorted_speed` can be approximated with the interval $[\texttt{cells}[\text{cell\_id}_L - 2], \texttt{cells}[\text{cell\_id}_U + 3] - 1]$. The region of buffers `cells` touched by the atomic operation can also be approximated with the interval $[\text{cell\_id}_L - 2, \text{cell\_id}_U + 2]$ as for kernel *count*.

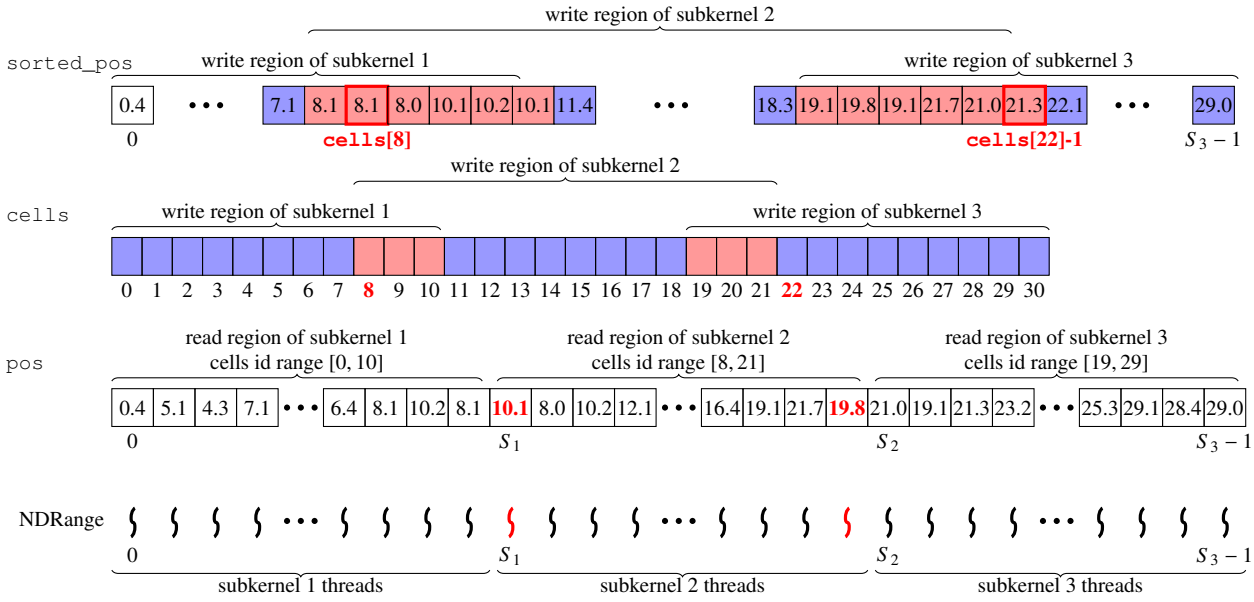


Figure 3.13: Partitioning arrays from the *sort* kernel.

The developer of the application can annotate the kernel with these intervals as shown lines 15 to 19 in the code. Thus, to compute the region of buffer `cells` touched by a sub-kernel of sub-NDRange $[S_1, S_2 - 1]$, only the two elements at indices S_1 and $S_2 - 1$ are read from buffer `pos` as shown in **Figure 3.13**. In this example the values of these two elements are 10 and 19, hence the region of buffer `cells` touched by this sub-kernel is computed as the interval $[8, 21]$. Finally, to compute the region of buffers `sorted_pos` and `sorted_speed` written by this sub-kernel, only the two elements at indices 8 and 22 are read from buffer `cells`. This region is computed as $[a, b - 1]$ where a and b are the values of buffer `cells` at indices 8 and 22.

In this kernel, the result of the atomic increment into buffer `cells` performed by a thread t_1 may be read by a thread $t_2 \neq t_1$. This is the case when t_1 and t_2 are in charge of atoms belonging the same cell. When splitting this kernel into sub-kernels executed onto different devices, it is necessary that given a thread t reading buffer `cells` at location l, all thread t' modifying atomically the value of buffer `cells` at this location are executed on the same device to ensure correctness. To tackle

this issue, our analysis automatically detects that the result of the atomic operation line 9 in the code is used inside the kernel. The region touched by this atomic operation is then considered as a write-region as explained in **Section 3.1.4**.

For this kernel, the memory region analysis of buffers `sorted_pos` and `cells` results in non-disjoint write-region as illustrated in **Figure 3.13**. In this case, the overlapping write-regions are considered as *may-write regions* (highlighted in red in the figure) and non-overlapping write-regions as *must-write regions* (highlighted in blue in the figure). The analysis is not capable of determining on which device the *may-write* regions will be written but this issue is resolved by computation replication. The sub-NDRange of each sub-kernel is increased until contiguous *must-regions* cover the whole buffers as explained in **Section 3.1.3**.

3.2.3 Evaluation

We evaluate in this section the performance obtained when partitioning SOTL automatically with our method compared to the performance obtained with an optimized version split by hand.

Methodology: Experiments are conducted on the CONAN platform which has 3 NVIDIA Tesla M2075 GPUs. We measure the total execution time of the application when computing 100 iterations of the simulation varying the size of the input configuration from 100k to 20000k atoms. In order to only evaluate the efficiency of our data partitioning method, the input configurations correspond to a regular workload without any load variation. Load balancing issues arising when executing this application onto heterogeneous devices with an irregular input configuration are tackled in the next chapter.

We compare the performance obtained for 3 different versions:

- ORIGINAL: corresponding to the execution of the original version of SOTL on a single GPU.
- AUTO: corresponding to the execution of the version of SOTL partitioned automatically on multiple GPUs using our method.
- HAND: corresponding to the execution of the version of SOTL split by hand on multiple GPUs.

The HAND version of SOTL is an optimized version for multi-device execution overlapping communication and computation. In this version, the domain of the simulation is split into several sub-domains, one per device. At each iteration, each device only sorts atoms contained in its sub-domains. Then, 3 kernels are executed to compute the forces: *border_left*, *border_right* and *center*. The two first kernels compute the forces only for atoms belonging to the cells at the borders of the device sub-domain. The last kernel computes the forces for all atoms of the device sub-domain except those belonging to the border cells. While executing this kernel, borders are exchanged between devices in order to overlap communication and computation.

Results: **Figure 3.14** shows the speedups obtained by the AUTO and HAND versions on 2 and 3 GPUs compared to the performance of the ORIGINAL version on a single GPU.

We can see that the AUTO version obtained poor performance with small input configurations. Indeed, even with our memory region analysis reducing the amount of data to transfer between

kernels, computation and communication are not overlapped. Hence, the cost of the data transfers is too high to obtain speedups when there is not enough computation.

However, from an input size of 10000k, the AUTO version obtained very good performance with speedups of 1.73X on 2 GPUs and 2.39X on 3 GPUs. With an input size of 20000k the AUTO version obtained even better performance, with a speedup of 1.94X on 2 GPUs and a speedup of 2.73X on 3 GPUs.

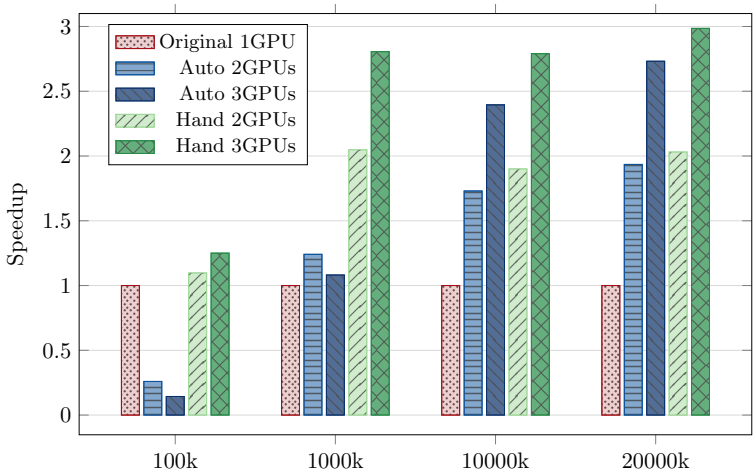


Figure 3.14: Performance obtained when partitioning SOTL on multiple GPUs.

The HAND version gives an upper limit of the maximum performance achievable when partitioning this application on multiple GPUs. We can see that the HAND version obtained very good speedups from an input size of 1000k. Nonetheless, from 20000k atoms the performance obtained with our method are very close to the performance obtained with the HAND version.

Automatically partitioning this complex application onto multiple devices with our method only requires to add a few pragmas to the original version of SOTL. On the other hand, starting from the ORIGINAL single-device version of SOTL, tremendous coding efforts are necessary to obtain the optimized HAND version.

3.3 Related Work

Many works propose techniques to distribute the workload of OpenCL kernels to multiple devices by splitting the NDRange at workgroup granularity as we do. However, few of them aim at automatically partitioning the data of complex kernels containing indirect memory accesses and atomic operations and minimizing the amount of data to transfer. Most of them require simple memory access patterns in order to split the data and do not keep track of the memory regions of buffers owned by each device in order to avoid redundant data transfers in the context of iterative applications with multiple kernels.

The Maat library [43, 51, 52] provides an interface resembling the OpenCL API with new abstractions such as super context, super buffer and super kernel to manage the co-execution of OpenCL kernels onto multiple devices. A super context can contain all devices from the system and is not limited to a single vendor (e.g. Intel or NVIDIA) as in OpenCL. Super kernels and super buffers are kernels and buffers that are valid across all the devices that are contained in a super

context. The creation of a super buffer results in the creation of a buffer of the same size for all the devices as in our method. Two new commands are added to the OpenCL API in order to write data to the devices. The *clWriteSuperBuffer* command broadcasts the data to all devices. On the contrary, with the *clDelayedWriteSuperBuffer* command, the data transfer is performed just before the kernel execution and each device only receives the data it needs to execute its sub-kernel. However, in order to split the data their method only support kernels with regular access patterns. Moreover, they only handle kernels where each thread only produces the result for the index of the output buffer corresponding to its index in the NDRange.

Kofler *et al.* [53] present a method for OpenCL task partitioning. Using a combination of source-to-source transformation and static code analysis they automatically translate a single-device OpenCL application into a multi-device application. Their analysis collect the access patterns of each buffer and checks if the access expression is (a) a constant, (b) the result of a convex function depending on the thread id, or (c) something else. If only accesses of type (b) occur, the buffer is split among all devices. For input buffers, if accesses of type (a) or (b) occur the entire buffer is broadcasted to every device. For output buffers, only accesses of type (b) are handled. For all others access patterns their method is not capable of partitioning a kernel.

In [40], the way to partition the data relies on a sampling run of the kernel to determine the buffer access ranges for each candidate partition. However, their method requires that array indices are an affine function of thread ids (i.e. global ID, work-group ID and local ID), and does not handle array reference in conditional statements. After a partitioning of the kernel has been determined, the runtime allocates on each device a buffer whose size is exactly the same as that of the access range of the sub-kernel it executes. Thus, their method makes it possible to execute kernels whose memory consumption exceeds the memory capacity of a single device. In order to distribute the workload, they propose six different ways of partitioning a NDRange with three dimensions. The three first partitionings correspond to splitting only one of the three dimensions as we do. The three others consist in splitting two dimensions of the NDRange in a way that all work-groups allocated on the same device have contiguous work-group IDs in each dimension. The runtime then selects the partitioning of the NDRange that minimizes the amount of data to transfer.

Sakai *et al.* [54] extend work in [40] with a data decomposition method for multi-dimensional data that cannot be entirely stored in the GPU memory and aiming at accelerating a single-GPU code on a multi-GPU system. This method uses a sample run and is limited to kernels whose memory references are given as affine functions of the thread and thread block indices.

In [55], Kim *et al.* propose a method to partition OpenCL kernels across different devices in a cluster. However, mapping tasks and data to these devices is left to the programmer.

Li *et al.* [37] present STEPOCL, a tool which takes as input kernels along with a configuration file and generates automatically an OpenCL multi-devices application. The configuration file describes how to split data, the control flow of the program, and allows to have specialized kernels for different architectures. Their method makes it possible to execute OpenCL kernels with large workloads that do not fit in the memory of a single device.

Pandit *et al.* present FluidiCL [56], a framework that allows the co-execution of an OpenCL kernel on a CPU and a GPU. In order to split the data, input buffers are broadcasted to both devices. Then the partial output computed by each device is merged using a "merge" kernel executed on the GPU. This merging is carried out by maintaining a copy of the unmodified buffer (the original buffer) and comparing it with the data computed on the CPU. If the data computed on the CPU differs from the original buffer, it is copied to the GPU buffer. Nevertheless, the proposed approach cannot be easily generalized for any number of devices and the code transformation is not fully

automated.

The SKMD framework [41] uses a flattened view of N-dimensional work-groups in order to distribute the workload of an OpenCL kernel to multiple devices. A static analysis of each kernel is performed in order to compute the buffer memory access range of each work-group. They differentiate between contiguous kernels, in which each of the threads writes the result in contiguous locations and discontiguous kernels. If for each buffer, the function of index where the buffer is accessed is an affine function of the work-item IDs, the kernel is considered as a contiguous kernel. For contiguous kernels only the necessary data is transferred back and forth from each device once the partitioning decision is made. On the other hand, for discontiguous kernels the entire input is transferred to each device and the output must be merged. In order to merge the output from different devices, they generate a "merge" kernel which is executed on the CPU. The "merge" kernel reuses the original kernel function after removing the computational part and replacing the values stored to output buffers with the values computed on the GPU. This way the "merge" kernel is executed with the NDRange of the sub-kernel executed on the GPU and only the locations touched by the GPU kernel will be copied to the CPU buffer. However, as many merge kernels as there are GPUs must be executed in order to merge the partial output computed by all devices. Moreover, when the computational part of the kernel is used to compute the index where the output buffer is written, it cannot be removed from the "merge" kernel. In this case, the execution time of the "merge" kernel would be the same as the execution time of the original kernel.

None of these works is capable of partitioning complex applications with spatial binning such as the SOTL application presented in the previous section. For input buffers containing indirect memory accesses as in SOTL, the only solution they propose would be to broadcast the whole input buffers to every device. This would result in a huge overhead for this application. Moreover, most of them are not capable of merging the partial output from different devices when the locations of the elements are given as indirect memory accesses as in the *sort* kernel from SOTL. Even the merging techniques proposed in FluidiCL [56] and in SKMD [41] cannot cope with kernels containing atomic operations as in the *count* kernel from SOTL.

Finally, in VAST [57], the authors propose a method partitioning the data parallel workload of an OpenCL kernel. This method does not target the execution of the kernel on multiple devices but aims at virtualizing memory space of the GPU. To overcome the limited size of the physical memory of the GPU their method performs execution of subsets of work-groups consecutively. VAST applies the inspector-executor model to generate a new type of page table accessible for the OpenCL kernels. Their method extracts the precise working set required for the divided workload and supports kernels with indirect memory access pattern. The solution they propose is not limited to indirect memory accesses where the values of the indirection array are increasing as in our method.

3.4 Summary

Heterogeneous multi-devices architectures are very complex to program. Many efforts remain to be done to improve the programmability of these architectures in order to achieve exascale. To express the parallelism of their applications, developers should be free from issues related to the underlying architecture where their applications will be executed.

To that end, OpenCL propose an abstraction that allows to execute the same code on different architectures (e.g. CPUs, GPUs). However, when targeting the execution of an OpenCL application

onto multiple devices the developer still has to partition the workload and manage the data transfers between devices by hand. Adapting a single-device application to multi-device architectures is a tedious and error prone task.

In order to ease the development of applications for multi-device architecture, we proposed in this chapter an automatic data partitioning method. Contrary to existing related works, the method we propose allows to automatically partition applications with complex kernels that may contain indirect memory accesses and atomic operations. By calculating precisely the region of buffers read and written by each sub-kernel and keeping track of the region of buffers owned by each device, our method limits the amount of data to transfer between devices.

However, once the application can be automatically partitioned onto to multiple devices, several issues still need to be addressed. When partitioning an application onto heterogeneous devices, load balancing issues may arise if the devices exhibit different compute capabilities, even if the workload of the application is regular. In addition, the communication time between devices must be taken into account to minimize the total execution time of the application. Moreover, some applications may have irregular workloads and in the case of iterative applications there can be load variations over iterations. Addressing these issues is the subject of the next chapter.

CHAPTER 4

Dynamic Load Balancing of Iterated Sequences of Irregular Kernels

5.1 Objectives and Principles 100

5.1.1 PARCOACH 100

5.1.2 Objective of our Full-Interprocedural Analysis 107

5.2 Full-Interprocedural Analysis 107

5.2.1 PPCFG Construction 107

5.2.2 Collective Error Detection 108

5.3 Code Instrumentation 109

5.4 Experimental Results 111

5.4.1 Static Analysis Results 111

5.4.2 Execution Results 114

5.5 Summary 115

One of the main challenges when adapting a single-device application to multiple heterogeneous devices is to automatically balance the load between devices so as to minimize the total execution time of the application. The type of applications we consider in this chapter consists in iterated sequences of irregular kernels. The adaptation of our method to non-iterative applications is discussed in **Section 7.1.2**.

As explained in **Section 2.2.2**, many factors must be taken into account in order to minimize the total execution of an iterative irregular application. The load balancing method must take into account the heterogeneity of the hardware, but also the possible irregularity of workload and the possible load variations between repeated executions of the kernel sequence. In addition, the method has to take into account the communication times induced by the partitioning of each kernel in order to minimize the overall execution time of the application.

This chapter presents our automatic method to dynamically balance the workload of an iterative application on heterogeneous architectures. This method corresponds to the implementation of the *split* function presented in **Figure 2.5b, page 30**. The method we propose is purely dynamic and does not require offline profiling nor sampling of the application in order to balance the workload.

4.1 Load Balancing Computation

This section presents our approach to automatically balance the load between the devices without any prior knowledge of the workload of the application. In this section we only focus on minimizing the execution time of each kernel. We do not consider the impact of the partitioning of each kernel on the volume of data to transfer between kernels in the case of multi-kernel applications. The principle of our method consists in partitioning each kernel with a uniform partitioning for the first iteration (same fraction of the original iteration space on each device). Then, for each of the following iterations the partitioning is refined based only on the sub-kernels execution times at the previous iteration. The method we propose handles kernels with irregular workload and is able to cope with load variations between repeated executions of the same kernel.

4.1.1 Formalization

Given a kernel and n devices, the problem consists in determining how to distribute the parallel iteration space of the kernel among the devices so as to minimize its execution time. Each device executes the same kernel, but possibly with a different number of work-groups and different data (referred to as sub-kernel).

Work-groups are indexed in OpenCL by a vector of indices among a rectangular space (from 1D to 3D) called the NDRange. Selecting a partitioning for a kernel boils down to defining a subvolume of indices for each device. The subvolumes we consider are obtained by selecting one smaller interval in one dimension of the NDRange.

The *offset* on each device is the first index of this interval and the *partitioning ratio* defines the size of this interval. We formally define the partitioning ratio as a value x_i in $[0, 1]$ corresponding to the ratio between the number of work-groups allocated to the device i and the total number of work-groups ng of the kernel. The number of work-groups ng is only known at runtime when the kernel is launched with a given NDRange.

We define $f_i(x_i, \text{offset}_i, t)$ as the mean time to execute one work-group on device i, when a sub-kernel of partitioning ratio x_i is executed at time step t, with an offset offset_i. The total execution time of this sub-kernel is therefore $f_i(x_i, \text{offset}_i, t) * x_i * ng$.

The solution to the problem consists in finding the time T and the partitioning ratios x_i and the offsets offset_i such that the system in **Figure 4.1a** is fulfilled.

$$
\begin{array}{ll}
\min T & \\
f_1(x_1, \text{offset}_1, t) * x_1 * ng & \leq T \\
\dots & \\
f_n(x_n, \text{offset}_n, t) * x_n * ng & \leq T \\
\sum_i x_i = 1 &
\end{array}
$$

(a) Initial formulation.

$$
\begin{array}{ll}
\min T & \\
f_1(x_1, t) * y_1 * ng & \leq T \\
\dots & \\
f_n(x_1 \dots, x_n, t) * y_n * ng & \leq T \\
\sum_i x_i = 1, & \sum_i y_i = 1
\end{array}
$$

(b) Generalized formulation.

Figure 4.1: Formulations of the partitioning problem.

The functions f_i are not known precisely, but they can be determined at runtime for a given x_i, offset_i and t by measuring the execution time of a sub-kernel of partitioning ratio x_i and offset offset_i at iteration t.

We arbitrarily order the offsets by increasing id of device. Thus, with offsets defined by values in $[0, 1]$, $\text{offset}_1 = 0, \ldots, \text{offset}_n = \sum_{k<n} x_k$ and f_i no longer depends on offset_i but on all x_j such that $j \leq i$. We generalize this formulation by introducing a new set of variables, $y_i \in [0, 1]$, as shown in **Figure 4.1b**.

The optimal load balancing occurs when all sub-kernels take the same time to execute. Hence, it is obtained when $f_i(x_1, \ldots, x_i, t) * y_i * ng = T$ for all i.

Thus, $y_i = \dfrac{T}{f_i(x_1, \ldots, x_i, t) * ng}$ with $\sum_i y_i = 1$ and $\sum_i \dfrac{T}{f_i(x_1, \ldots, x_i, t) * ng} = 1$ implies:

$$T = \frac{ng}{\sum_i 1/f_i(x_1, \ldots, x_i, t)}.$$

Now it is possible to define a function F such that $F_t(\mathbf{x}) = (\mathbf{y})$, with $\mathbf{x} = (x_1, \ldots, x_n)$ the vector of all x_i and $\mathbf{y} = (y_1, \ldots, y_n)$ the vector of all y_i, satisfying conditions from **Figure 4.1b**:

$$F_t(\mathbf{x}) = \left(\frac{T}{f_i(x_1, \ldots, x_i, t) * ng} \right)_i,$$

with:

$$T = \frac{ng}{\sum_i 1/f_i(x_1, \ldots, x_i, t)}.$$

The evaluation of $F_t(\mathbf{x})$ requires $O(n)$ basic arithmetic operations with n the number of devices. A solution to the problem of **Figure 4.1b** can be found by computing a fixed point of the function F: $F_t(\mathbf{x}) = \mathbf{x}$ or similarly, by finding the 0 of the function G_t: $G_t(\mathbf{x}) = \mathbf{x} - F_t(\mathbf{x})$

4.1.2 Resolution Method

First assume the function F_t does not depend on t, the iteration count. Several methods have been proposed in the literature for solving such problem, when the function is not known analytically:

- The fixed point method consists in computing the suite of partitioning ratios vectors $\mathbf{x}_k = F(\mathbf{x}_{k-1}), k \geq 1$ from some initial value $\mathbf{x}_0$. The evaluation of $F(\mathbf{x}_{k-1})$ requires to execute the kernel with the partitioning ratios $\mathbf{x}_{k-1}$. When the suite converges, it converges linearly towards a vector of optimal partitioning ratios satisfying the initial problem and achieving perfect load balance. The convergence depends on F and on the initial value $\mathbf{x}_0$ but is in general linear.

- The secant method, or its generalization for n-D space the Broyden's method [58], uses an approximate gradient to converge to the 0 of a function with a near quadratic convergence rate.

We implemented both methods to refine the partitioning ratio assigned to each device. F is evaluated at each kernel instantiation and provides the input necessary to instantiate the partition-ready kernel.

Finally, when the functions f_i also depend on the iteration count t, the fixed point equation becomes $F_{t-1}(\mathbf{x}_{k-1,t-1}) = \mathbf{x}_{k,t}$ with F changing for each term of the suite. For real applications, a good approximation of the solution at step t remains a good approximation at step $t + 1$. As the fixed point method converges quickly when the approximation is close to the solution, we believe this approach can be used for many real cases. We demonstrate for an N-Body application with

irregular workload that the fixed point method is able to stay close to the optimal, even when the optimal partitioning is varying with the iteration count (dynamic workload).

4.1.3 Evaluation

This section presents the evaluation methodology and result for our method.

Methodology: Experiments are conducted on two platforms:

- CONAN: 16-core Intel Xeon E5-2650 2.00GHz with 64GB, and 3 NVIDIA Tesla M2075 GPUs
- HAPPYCL: 12-core Intel Xeon E5-2680 2.80GHz with 64GB, 1 NVIDIA Tesla K20c GPU and 1 NVIDIA Quadro K5000 GPU

We evaluate our method on many benchmarks from various sources: AESEncrypt and MonteCarlo from the AMD SDK [59], EP from the SNU NPB Suite [47], a sparse matrix vector multiplication (SPMV), 9 benchmarks from the Polybench [60] ; and on an N-Body application: OTOO [48].

app	# kernels	workload	iterative	data partitioning	
				input	output
AESEncrypt	1	regular	repeated	split	split
EP	1	regular	iterative	split	split
MonteCarlo	1	regular	repeated	split	split
SpMV	1	irregular	iterative	broadcast	split
2DCONV	1	regular	repeated	split	split
2MM	2	regular	repeated	split	split
3MM	3	regular	repeated	split	split
GEMM	1	regular	repeated	split	split
GESUMMV	1	regular	repeated	split	split
Jacobi1D	2	regular	iterative	split	split
Jacobi2D	2	regular	iterative	split	split
SYRK	1	regular	repeated	split	split
SYR2K	1	regular	repeated	split	split
OTOO	1	dynamic	iterative	broadcast	split

Table 4.1: Applications and Benchmarks Description.

Benchmarks characteristics are presented in **Table 4.1**. The workload of the Sparse Matrix Vector Multiply kernel (SpMV) is irregular: In the chosen sparse matrix, rows with a high index have more non-zero elements than those with a low index. The workload of OTOO is dynamic: The input set corresponds to a non-uniform distribution of the masses in space. As this space is partitioned among the work-groups, this results in a non-homogeneous load distribution among the work-groups, changing with iteration number. All other benchmarks have regular workloads.

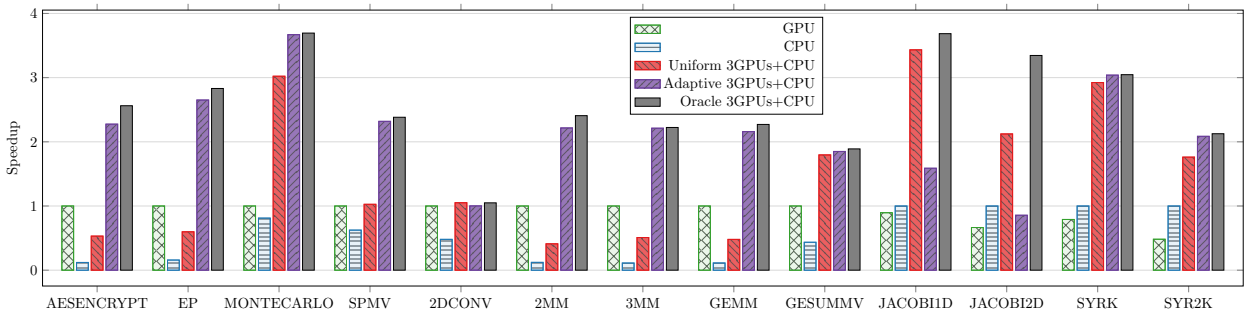
All benchmarks were repeated 100 times, and we measured the total execution time of the 100 iterations including data transfer times. For iterative application, the input data is transferred from the host to the devices only at the first iteration, then for the following iterations only the minimum data transfers between devices are performed. Finally, the output data is transferred from the devices back to the host only after the last iteration. On the other hand for non iterative

benchmarks, the input data is transferred from the host to the devices before each iteration and the output data is transferred from the devices back to the host after each iteration.

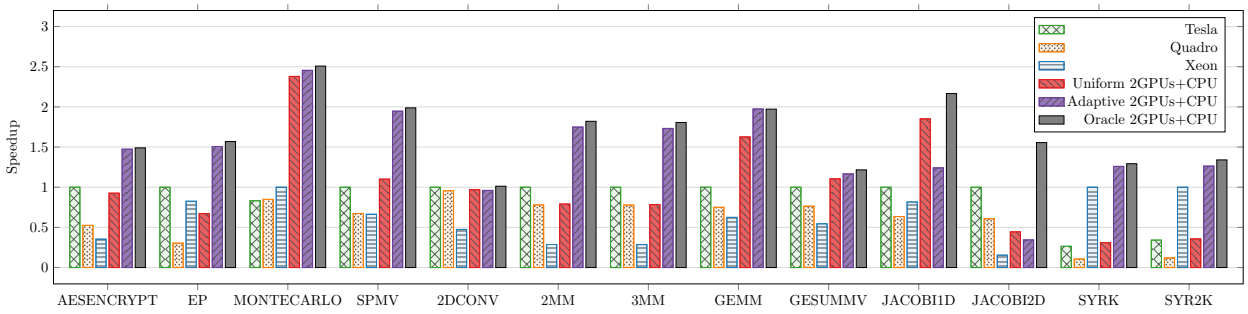
Table 4.1 shows whether the input data is split or broadcasted to the devices. For the SpMV kernel, the same input matrix is used for the 100 iterations and at each iteration the kernel computes the multiplication of the input matrix with a different vector. Hence, the input matrix is broadcasted to all devices only at the first iteration. For the OTOO kernel, atoms are stored into an octree and each device requires the whole octree to compute the forces applied to the subset of particles for which it calculates the displacement. Hence, the whole input is broadcasted to all devices. For all other benchmarks our method was able to split the input data and only the required data is transferred to each device.

Overall Speedups: **Figure 4.2** presents speed-ups compared to the best single-device performance on the two target architectures, for a large number of benchmarks when they are repeated 100 times. Speedups are shown for 3 different strategies:

- UNIFORM: When the partitioning ratio of each sub-kernel is $1/n$, with n the number of devices.
- ADAPTIVE: When the load is automatically adapted with our method.
- ORACLE: With this strategy, each kernel is directly partitioned with the partitioning ratios found after convergence of the ADAPTIVE method. For Jacobi1D and Jacobi2D, the ORACLE is obtained by giving the same partitionings for both kernels.



(a) CONAN.



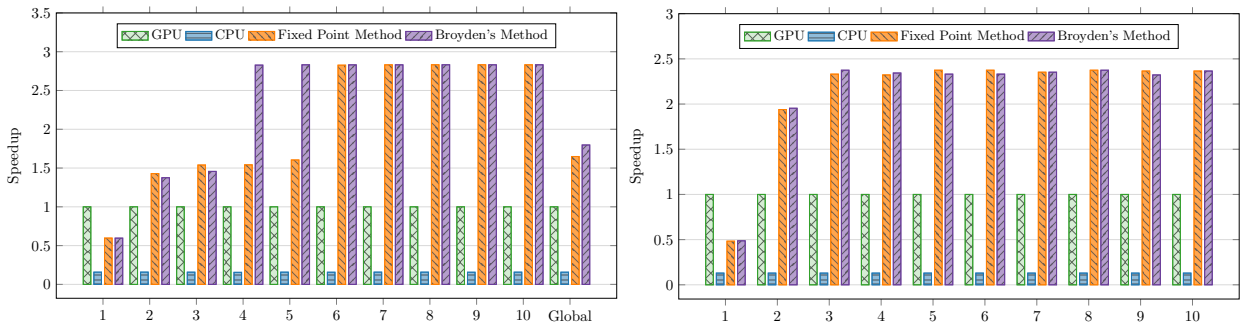
(b) HAPPYCL.

Figure 4.2: Performance of AESEncrypt, EP, MonteCarlo, OTOO, SPMV and some Polybench on CONAN and HAPPYCL. Original codes run only on one device. UNIFORM and ADAPTIVE are using sub-kernels automatically obtained by our method.

We observe the results of our method (ADAPTIVE) are close to the optimal, obtained when launching the kernel directly with the partitioning ratios obtained after convergence (ORACLE).

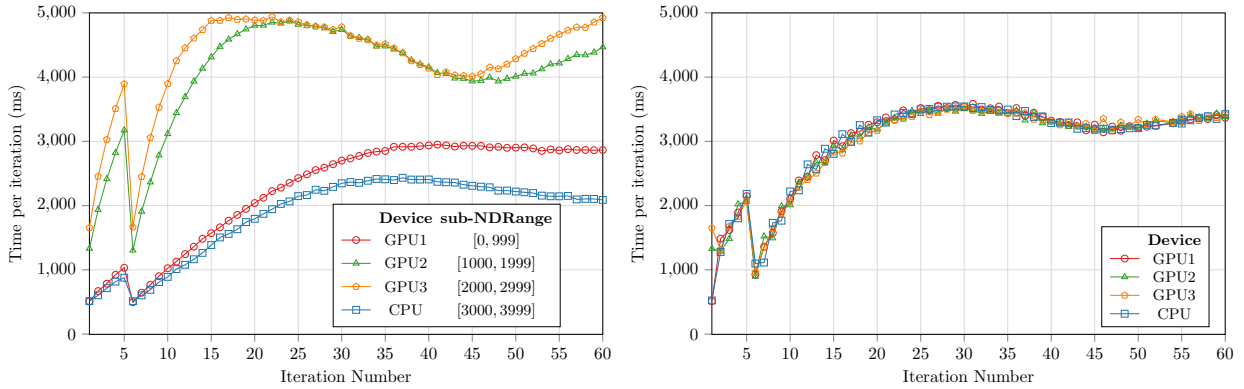
For Jacobi1D and Jacobi2D, the gap is more important because these benchmarks consist in 2 kernels, one stencil and one copy. Defining the same partitioning for both copy and stencil minimizes communication time as shown in **Figure 2.17a, page 42**. This method handles only one kernel at a time, and does not find the same partitioning for both kernels. These examples show the limitation of this method that does not take into account the communication times induced by the partitioning of each kernel in the case of multi-kernel applications.

Detailed Load Balancing: **Figure 4.3** shows the speed-up obtained on CONAN with AESEncrypt and EP compared to the best single-device performance, when these kernels are repeated 10 times. The speed-up shown here are per iteration. For the first iteration, the partitioning ratio is the same for the 4 devices (uniform hypothesis), explaining poor performance compared to the GPU performance. For EP, **Figure 4.3a**, the fixed point method requires 6 iterations to reach a maximum speed up of 2.8, whereas the Broyden's method converges in only 4 iterations leading to a better global speedup (shown in **Figure 4.3a**). For AESEncrypt, **Figure 4.3b** shows there is no such difference and both methods are similar, reaching a peak speed-up of 2.15 in 3 iterations only.



(a) EP. **(b)** AESEncrypt.

Figure 4.3: Speedup per iteration of EP and AESEncrypt.



(a) UNIFORM. **(b)** ADAPTIVE.

Figure 4.4: Performance of OTOO executed on CONAN (3GPUs+CPU) for 60 iterations.

Figure 4.4 shows how our method behaves when the load changes over 60 iterations. **Figure 4.4a** illustrates the time taken by each sub-kernel for OTOO when the partitioning ratio is the same for the 4 devices (Uniform strategy). From one device to the other, the execution time differs by more than a factor 3 (iteration 15 for instance). **Figure 4.4b** shows how the same load is shared among the four devices when it is continuously adapted by our technique (Adaptive strategy). As the 4 plots are close to each other, this shows the execution time is nearly optimal. We observe that convergence to the optimal only requires 2 iterations.

4.2 Communication-Aware Load Balancing

This section defines an improvement of the method to determine how to partition an iterated sequence of m kernels onto n devices. Unlike the method presented in the previous section, this new method takes into account both communication and computation times to balance the load. The partitioning of each kernel in the sequence is computed by solving a linear system at each iteration. At the end of iteration t, the linear system computes the partitioning ratios of each kernel in the sequence, in order to minimize the overall execution time of iteration $t + 1$. This linear system takes into account both communication and computation times and is based on previous iteration measures.

We first extend the previous formulation to an iterated sequence of multiple kernels by using a linear system. Solving this system is referred to as the Adaptive w/o Comm strategy. Then we present a new formulation of the problem that takes into account both computation and communication time to balance the load. Solving the linear system corresponding to this new formulation is referred to as the Adaptive w/ Comm strategy.

4.2.1 Formalization

The Adaptive w/o Comm Strategy

The Adaptive w/o Comm strategy finds partitioning ratios for each kernel from the sequence individually in order to minimize their execution times. This strategy does not take into account the transfer times between kernels induced by their respective partitionings. In this case, the linear system only relies on the execution times of the sub-kernels at iteration t to determine the partitioning ratios for iteration $t + 1$.

Given a sequence of m kernels and n devices, this strategy consists in finding the execution times $T^1, \ldots, T^m$ of each kernel, and the new partitioning ratios y_d^k of each kernel k on each device d for iteration $t+1$ such that the following system is fulfilled:

$$\begin{cases} \min T^1 + \ldots + T^m \\ \forall k = 1..m : \\ \quad \forall d = 1..n : \\ \quad\quad f_d^k(x_1^k, \ldots, x_d^k, t) * ng_k * y_d^k \leq T^k, \\ \quad\quad \sum_d y_d^k = 1 \end{cases}$$

This linear system is a generalization of the formulation presented in the previous section to a sequence of m kernels. The *partitioning ratio* for a kernel k and device d is a value x_d^k in $[0, 1]$ (with $\sum_{d=1}^{n} x_d^k = 1$) corresponding to the ratio between the number of work-groups allocated to the

device d and the total number of work-groups (ng_k). ng_k is known when kernel k is called. We define $f_d^k(x_1^k, \ldots, x_d^k, t)$ as the mean time to execute one work-group on device d at iteration t, when sub-kernels on devices $1, \ldots, d$ have respectively partitioning ratios $x_1^k, \ldots, x_d^k$. The execution time of the sub-kernel of k on device d is $f_d^k(x_1^k, \ldots, x_d^k, t) * ng_k * x_d^k$ and the total execution time of kernel k is $\max_{d=1,n}(f_d^k(x_1^k, \ldots, x_d^k, t) * ng_k * x_d^k)$. The functions f_d^k are not known precisely, but we determine the value of f_d^k with the execution time of the sub-kernel of k on device d at iteration t.

The Adaptive w/ Comm Strategy

The Adaptive w/ Comm strategy takes into account the data transfer times induced by the partitioning of each kernel from the sequence in order to minimize the overall iteration time. We model the volume of data to transfer between two data-dependent kernels as a function of their partitioning ratios. At each iteration, the parametric read and write regions of all sub-kernels are instantiated and we can determine the value of this function for the current partitioning ratios. At runtime, using a linear regression, we compute for each pair of data-dependent kernels and for each device a coefficient giving the volume of data to transfer depending on the partitioning ratios of these kernels. Then, using these coefficients we add new constraints to the linear system presented in **Section 4.2.1** modeling the communication times between all data-dependent kernels.

The objective function to minimize becomes:

$$\min T^1 + \ldots + T^m + T^1_{H2D} + T^1_{D2H} + \ldots + T^m_{H2D} + T^m_{D2H}$$

where T^k_{D2H} and T^k_{H2D} are devices-to-host and host-to-devices transfer times before the execution of kernel k. For each kernel k and device d we add the two following linear constraints:

$$T^k_{H2D_d} \leq T^k_{H2D} \tag{4.1}$$

$$\sum_{h=1}^{m} a_d^{h,k} * S_d(y_1^h, \ldots, y_d^h, y_1^k, \ldots, y_d^k) * \Omega_d^{H2D} \leq T^k_{H2D_d} \tag{4.2}$$

where:

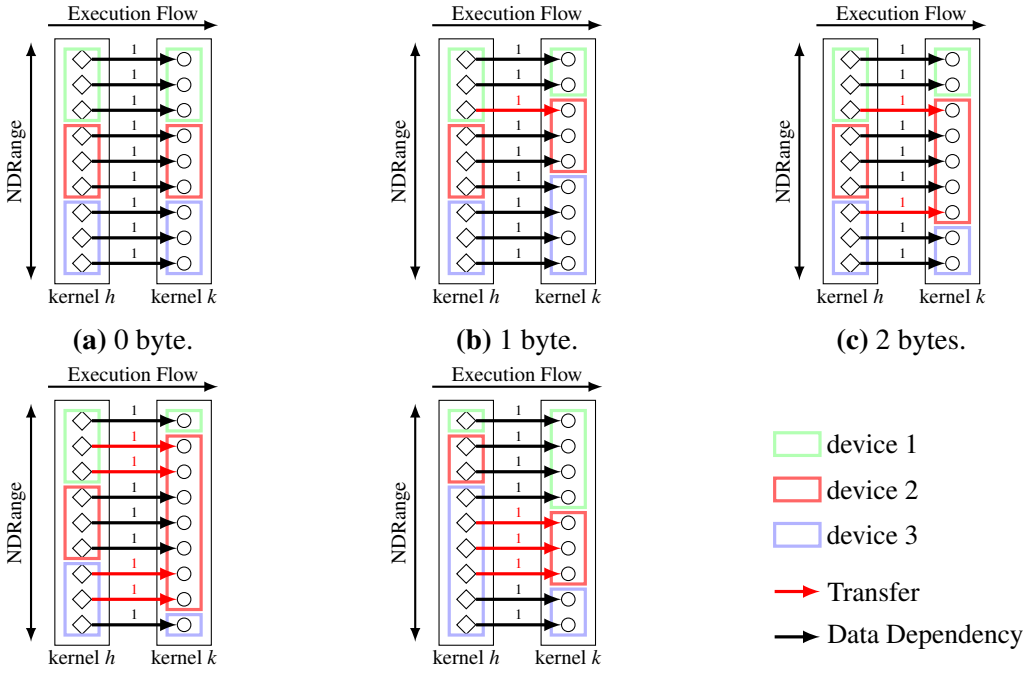
- the y_d^k are the unknowns of the system;
- $a_d^{h,k}$ and Ω_d^{H2D} are coefficients determined at runtime;
- S_d depends on the partitioning ratios y_d^k of kernel k at $t+1$ and on the partitioning ratios y_d^h of kernel h on which k depends.

The linear constraint (4.1) means that the transfer time from the host to the n devices before executing kernel k is equal to the longest host to device data transfer time (transfers to different devices are performed in parallel). The linear constraint (4.2) means that the host to device d transfer time before executing kernel k is the sum of the host to device d transfer times from each kernel h on which k depends. The meaning of the relation S_d and the coefficients $a_d^{h,k}$ and Ω_d^{H2D} are explained in the next paragraph.

Similar constraints are added for device to host transfers.

Let us now explain our communication modelization. When a kernel h writes to a buffer B that is read by a kernel k, data transfers may be required when those kernels are partitioned onto multiple devices. It is the case if a sub-kernel of k executed on device d reads a region of B that is written by a sub-kernel of h executed on another device. This data then comes from another device and a communication is required.

Figure 4.5 illustrates the amount of data to transfer between two dependent kernels h and k when they are partitioned onto 3 devices. The figure shows the amount of data to transfer to device 2 before executing kernel k for various partitionings of h and k. In this example, both kernel have the same NDRange of size 9, and each thread of k only depends on the 1 byte of data produced by the thread of h at the same position in the NDRange. Threads of kernel h are represented with diamonds and threads of kernel k with circles. The amount of data to transfer to device 2 depends on the partitioning of h and k. For example, in **Figure 4.5a** threads of k executed on device 2 only depend on the data produced by threads of h executed on the same device. Hence, no data transfer to device 2 is required before executing kernel k. On the contrary, in **Figure 4.5b** a thread of kernel k executed on device 2 depends on the 1 byte of data produced a thread of h executed on device 1. Hence, 1 byte must be transferred from device 1 to device 2 before executing kernel k.



(a) 0 byte. **(b)** 1 byte. **(c)** 2 bytes. **(d)** 4 bytes. **(e)** 3 bytes.

Figure 4.5: Amount of data to transfer to device 2 before executing kernel k depending on the partitioning of h and k when these kernel are partitioned onto 3 devices.

Assuming that kernels h and k have the same NDRange of size N and that each thread from h (resp. k) writes (resp. reads) one byte from (resp. into) buffer B at the index corresponding to its id in the NDRange, the amount of data to transfer between h and k can be determined using their partitioning ratios.

The region of B written by the sub-kernel of h on device d is:

$$W_d(y_1^h, \ldots, y_d^h) = N * [y_1^h + \ldots + y_{d-1}^h, y_1^h + \ldots + y_d^h - \frac{1}{N}]$$

and the region of B read by the sub-kernel of k on device d is:

$$R_d(y_1^k, \ldots, y_d^k) = N * [y_1^k + \ldots + y_{d-1}^k, y_1^k + \ldots + y_d^k - \frac{1}{N}].$$

The data not present on device d before execution of the sub-kernel of k is defined by the region $R_d(y_1^k, \ldots, y_d^k) - W_d(y_1^h, \ldots, y_d^h)$. When the two regions overlap, the amount of data to transfer is:

$$\begin{aligned} S_d(y_1^h, \ldots, y_d^h, y_1^k, \ldots, y_d^k) &= N * max(\sum_{i=1}^{d-1} y_i^h - \sum_{i=1}^{d-1} y_i^k, 0) \\ &\quad + N * max(\sum_{i=1}^{d} y_i^k - \sum_{i=1}^{d} y_i^h, 0), \end{aligned}$$

otherwise the amount of data is simply $N * y_d^k$.

Example:

In **Figure 4.5b** kernels h and k have the same NDRange of size 9. For kernel h each device execute 3 threads while for kernel k devices 1, 2 and 3 execute 2, 3 and 4 threads respectively. Hence, $y_1^h = \frac{3}{9}$, $y_2^h = \frac{3}{9}$, $y_3^h = \frac{3}{9}$, $y_1^k = \frac{2}{9}$, $y_2^k = \frac{3}{9}$, $y_3^k = \frac{4}{9}$, and the region of B written by the sub-kernel of h on device 2 is:

$$W_2(\frac{3}{9}, \frac{3}{9}, \frac{3}{9}) = 9 * [\frac{3}{9}, \frac{6}{9} - \frac{1}{9}] = [3, 5]$$

and the region of B read by the sub-kernel of k on device 2 is:

$$R_2(\frac{2}{9}, \frac{3}{9}, \frac{4}{9}) = 9 * [\frac{2}{9}, \frac{5}{9} - \frac{1}{9}] = [2, 4].$$

In this case, R_2 and W_2 overlap, hence the amount of data to transfer to device 2 before executing k is:

$$S_2(\frac{3}{9}, \frac{3}{9}, \frac{3}{9}, \frac{2}{9}, \frac{3}{9}, \frac{4}{9}) = 9 * max(\frac{3}{9} - \frac{2}{9}, 0) + 9 * max(\frac{5}{9} - \frac{6}{9}, 0) = 1.$$

In **Figure 4.5e**, the partitioning ratios are: $y_1^h = \frac{1}{9}$, $y_2^h = \frac{2}{9}$, $y_3^h = \frac{6}{9}$, $y_1^k = \frac{4}{9}$, $y_2^k = \frac{3}{9}$, $y_3^k = \frac{2}{9}$. Hence, the region of B written by the sub-kernel of h on device 2 is $W_2 = [1, 2]$ and the region of B read by the sub-kernel of k on device 2 is $R_2 = [4, 6]$. In this case, R_2 and W_2 do not overlap. The amount of data to transfer to device 2 is therefore 3 bytes.

In real applications, two dependent kernels do not necessary have the same NDRange and threads from both kernels can write buffers at any location. We model the communication volume on device d from kernel h to kernel k as a function of their partitioning ratios $g_d^{h,k} = a_d^{h,k} * S_d$ where $a_d^{h,k}$ is a coefficient and S_d is over-approximated by always considering that W_d overlaps R_d. At each iteration the parametric regions of kernels h and k are instantiated and we know the value of $g_d^{h,k}$ for their current partitioning ratios. Hence, the coefficients $a_d^{h,k}$ are computed at runtime using a linear regression. Finally, the data transfer time from h to k is $a_d^{h,k} * S_d * \Omega_d$ where Ω_d is the time to transfer one byte from host to device d.

The full system is shown in **Figure 4.6** where: $a_d^{h,k}$ are coefficients computed for the host-to-devices constraints, $b_d^{h,k}$ are coefficients computed for the devices-to-host constaints, Ω_d^{H2D} is the time to transfer one byte from host to device d and Ω_d^{D2H} is the time to transfer one byte from device d to the host.

$$
\begin{cases}
\min T^1 + \ldots + T^m + T^1_{H2D} + T^1_{D2H} + \ldots + T^m_{H2D} + T^m_{D2H} \\
\forall k = 1..m : \\
\quad \forall d = 1..n : \\
\quad\quad f_d^k(x_1^k, \ldots, x_d^k, t) * ng_k * y_d^k \leq T^k, \\
\quad\quad \sum_d y_d^k = 1 \\
\forall k = 1..m : \\
\quad T^k_{H2D_d} \leq T^k_{H2D} \\
\quad \forall d = 1..n : \\
\quad\quad \sum_{h=1}^m a_d^{h,k} * S_d(y_1^h, \ldots, y_d^h, y_1^k, \ldots, y_d^k) * \Omega_d^{H2D} \leq T^k_{H2D_d} \\
\forall k = 1..m : \\
\quad T^k_{D2H_d} \leq T^k_{D2H} \\
\quad \forall d = 1..n : \\
\quad\quad \sum_{h=1}^m b_d^{h,k} * S_d(y_1^h, \ldots, y_d^h, y_1^k, \ldots, y_d^k) * \Omega_d^{D2H} \leq T^k_{D2H_d}
\end{cases}
$$

Figure 4.6: Linear system solved at each iteration with the Adaptive w/ Comm strategy.

The total number of constraints of the system is $4(m * n) + 2m$ with m the number of the kernels and n the number of devices. We show in the next section that the overhead induced by resolving the system at each iteration is negligible.

4.2.2 Evaluation

This section presents the evaluation methodology and result for our new method.

Methodology: Experiments are conducted on two platforms:

- conan: 16-core Intel Xeon E5-2650 2.00GHz with 64GB, and 3 NVIDIA Tesla M2075 GPUs
- happyCL: 12-core Intel Xeon E5-2680 2.80GHz with 64GB, 1 NVIDIA Tesla K20c GPU and 1 NVIDIA Quadro K5000 GPU

We evaluate our method on 3 regular benchmarks: Jacobi1D, Jacobi2D and FDTD2D; 1 irregular benchmark: 2SpMV; and 1 application with dynamic load variations: SOTL presented in **Section 3.2**. The Jacobi1D benchmark (2 kernels) consists in stencil kernel followed by a memcpy from the output buffer to the input. The Jacobi2D benchmark (2 kernels) is the 2-dimensional version of the Jacobi1D benchmark. FDTD2D consists in a succession of 3 stencil kernels. The 2SpMV benchmark (2 kernels) consists in a Sparse Matrix-Vector Multiplication applied on two different matrices, the output vector of one kernel is the input vector of the following one. Both kernels present irregular workload among threads, due to the sparsity structure of each matrix. For the SOTL application, we have tested our method on 2 configurations: SOTL St corresponds to a static regular workload whereas SOTL Dyn corresponds to dynamic irregular workload.

Benchmarks characteristics are summarized in **Table 4.2**. All benchmarks are iterative, hence input data is transferred from the host to the devices only at the first iteration, for the following iterations only the minimum data transfers between devices are performed and the output data is transferred from the devices back to the host only after the last iteration. The input data is split for all benchmarks except for 2SpMV were the 2 matrices are broadcasted to all devices at the first iteration.

app	# kernels	workload	iterative	data partitioning	
				input	output
Jacobi1D	2	regular	iterative	split	split
Jacobi2D	2	regular	iterative	split	split
FDTD2D	3	regular	iterative	split	split
2SpMV	2	irregular	iterative	broadcast	split
SOTL St	6	regular	iterative	split	split
SOTL Dyn	6	dynamic	iterative	split	split

Table 4.2: Applications and Benchmarks Description.

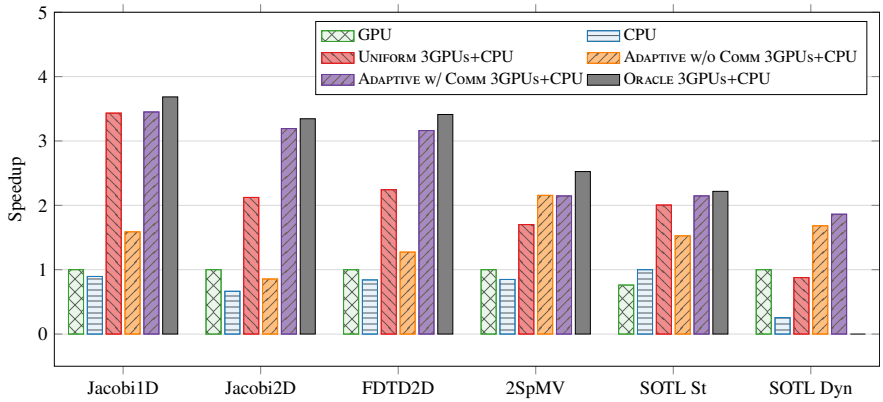
Overall Speedups: **Figure 4.7** presents speed-ups compared to the best single-device performance for 4 different strategies:

- UNIFORM: When the partitioning ratio of each sub-kernel is $1/n$, with n the number of devices.
- ADAPTIVE W/O COMM (cf. **Section 4.1**)
- ADAPTIVE W/ COMM (cf. **Section 4.2**)
- ORACLE: With this strategy, the kernels in the sequence are directly partitioned with the partitioning ratios found after convergence.

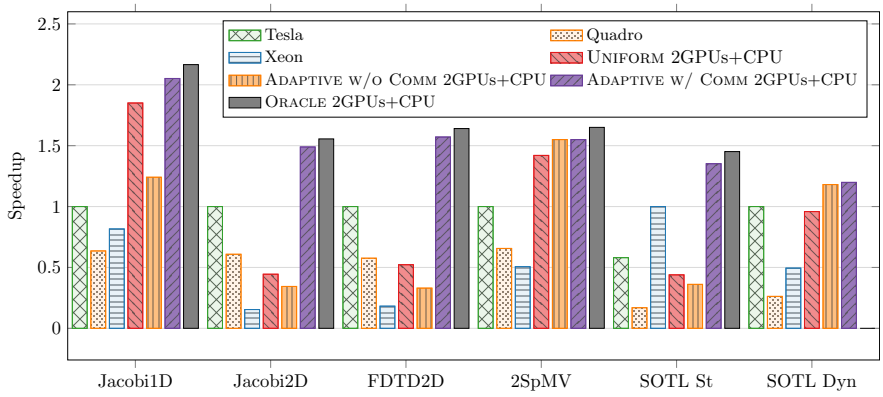
For the two adaptive strategies the kernel sequence is partitioned with a uniform partitioning at the first iteration, then the partitioning is continuously adapted by resolving a linear system after each iteration. Contrary to the ADAPTIVE W/ COMM strategy, for the ADAPTIVE W/O COMM strategy the linear system is resolved without the communication constraints. For SOTL Dyn, there is no ORACLE since the workload dynamically changes with iteration number and the solver never converges.

For the 4 benchmarks, we observe the results of the ADAPTIVE W/ COMM strategy are close to the optimal ORACLE strategy. The small difference of performance obtained with these two strategies shows that the overhead of resolving a linear system at each iteration is negligible. The ADAPTIVE W/O COMM strategy obtained poor performance for Jacobi1D, Jacobi2D and FDTD2D. Since this strategy only minimizes computation time, a slow down due to data transfers is observed. For these 3 benchmarks all kernels must have the same partitioning to avoid penalizing transfer times. For 2SpMV the same speedup is obtained with both adaptive strategies since the transfer times induced by the partitioning minimizing the computation time is negligible.

For SOTL St, the kernel used to compute the force were optimized for older GPUs, resulting in better performance on the CPU than the GPU on CONAN. We can see that the ADAPTIVE W/ COMM strategy obtains near optimal performance with SOTL St whereas the ADAPTIVE W/O COMM results in a slowdown on HAPPYCL. For this application, the time taken to compute one iteration is mostly taken by the *force* kernel. To avoid penalizing transfers, other kernels from the sequence must be



(a) CONAN.



(b) HAPPYCL.

Figure 4.7: Overall Results Obtained.

partitioned according to this kernel. This is achieved by the ADAPTIVE W/ COMM strategy. **Figure 4.8** demonstrates the effectiveness of the ADAPTIVE W/ COMM strategy which decreases the transfer time by a factor of 6. For SOTL Dyn, the workload varies at each iteration and the partitioning ratios minimizing the overall iteration time are different for each iteration. Thus, even with the ADAPTIVE W/ COMM strategy important data transfers are required each time the partitioning of the kernel sequence varies. Hence, the ADAPTIVE W/ COMM strategy has lower impact on performance for this version of SOTL.

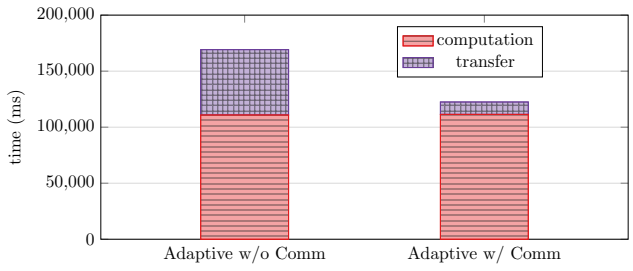
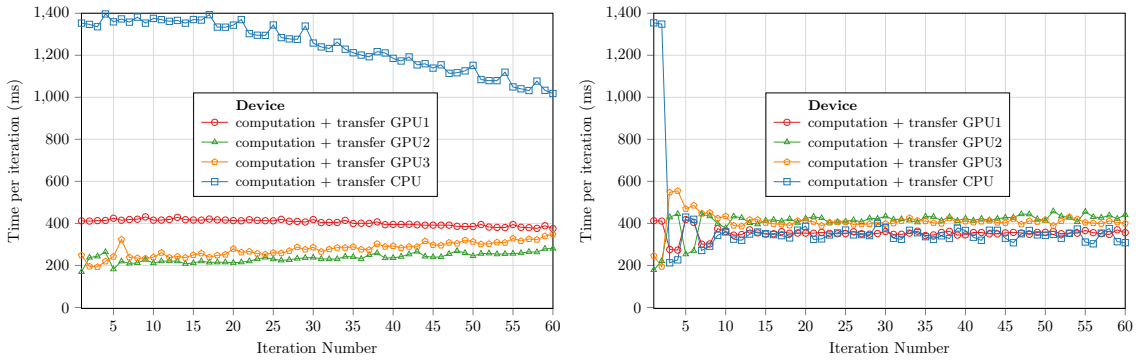


Figure 4.8: Time taken by computation and transfer when SOTL St is partitioned on CONAN using ADAPTIVE W/O COMM strategy versus ADAPTIVE W/ COMM strategy.



(a) UNIFORM. **(b)** ADAPTIVE W/ COMM.

Figure 4.9: Total time per iteration when SOTL Dyn is partitioned on CONAN.

Detailed Load Balancing: **Figure 4.9a** shows total execution time of the task chain for the first 70 iterations when SOTL Dyn is partitioned uniformly onto 3 GPUs and 1 CPU. The input set corresponds to a non-uniform distribution of particles in space. This results in a non-homogeneous load distribution among work-groups, changing with iteration number as the particles move. We can see on **Figure 4.9b** that the load balancing is nearly optimal with the ADAPTIVE W/ COMM strategy since the 4 plots showing the time per iteration on each device are close to each other.

This application requires all the mechanisms presented in **Chapter 3** to be partitioned onto multiple devices and in particular computation replication (cf. **Section 3.1.3**). The gap observed between iteration times of GPU2, GPU3 and GPU1,CPU is due to computation replication. Indeed, first and last devices (GPU1 and CPU) replicate computation only on one side of their sub-NDRange whereas GPU2 and GPU3 replicate computation on both sides.

4.3 Related Work

Many works propose techniques to partition the workload of data parallel kernels to heterogeneous multi-device architectures. These works usually target OpenCL applications, as OpenCL kernels can be executed onto different architectures. Most of them partition the workload at workgroup granularity as we do and are often referred as *coarse-grain partitioning* methods. These approaches can generally be classified with two types: static and dynamic. Static methods usually rely on static code analysis, offline profiling or training to partition the workload whereas pure dynamic methods partition the workload dynamically at runtime. A different approach is described by Magni *et al.* [61], where work-groups are fused in order to coarsen the granularity. The coarsening factor is determined by a machine learning technique, statically. Finally, other approaches specifically target irregular applications and propose workload reshaping methods where the workload is first reordered according to the amount of work per thread and then partitioned. These methods are often referred as *fine-grained partitioning* methods.

4.3.1 Static Coarse-grained Partitioning

Grewe *et al.* [62] propose a pure static task partitioning method without profiling of the target program on CPU-GPU systems. Their approach relies on machine-learning techniques to predict

the best partitioning between CPU and GPU and is based on static code features.

In [36] the authors present the Qilin system that automatically partitions threads to one CPU and one GPU by providing new APIs that abstract away two different programming models, Intel Thread Building Blocks and CUDA. Qilin relies on extensive off-line profiling to create a performance model for each task on each device. This information is then used to calculate a good partitioning across the devices.

Kofler *et al.* [53] present a method for OpenCL task partitioning relying on offline generated model. This model is based on artificial neural networks, relying on the features of the kernels, including their input sizes.

Seo *et al.* [63] propose an automatic work-group size selection technique for OpenCL kernels on multicore CPUs. Their method uses a profiling-based algorithm. Heterogeneity is not handled however and kernels are assumed to be work-group size independent.

The SKMD framework [41] relies on a performance prediction model to predict the execution time of an OpenCL kernel on each device and partition its workload. This performance predictor utilizes a linear regression model on profile data collected before the real execution. Then the partitioning is determined using a decision tree heuristic in order to take transfer cost and performance variances of devices into account. Nevertheless, SKMD does not handle kernels with irregular behaviors whose control flows depend on the values of the array parameters.

In [64], the authors propose to model workload distribution problem as a mixed-integer non-linear programming (MINLP) whose objective function minimizes the variance of execution times among GPUs. A performance model, built from training runs, and a reference run are required, as input to the MINLP solver. Their approach applies to only one task and does not take data transfer into account.

The solution presented in [65] to the problem of optimal distribution of the workload of data-parallel scientific applications on heterogeneous computing systems uses functional performance models (FPMs) of processing elements and FPM-based data partitioning algorithms. It may be difficult to balance the workload with this approach when there are multiple kernels.

All these works only focus on single kernel applications and are poorly suited for partitioning a sequence of kernels as they do not take into consideration the dependences between multiple kernels. Moreover, these works only target applications with regular workloads and cannot adapt to irregular workload and dynamic load variations. do not consider the impact the partitioning of one kernel

In MKMD [42], the authors extend SKMD [41] to applications to applications with multiple communicating kernels. The MKMD framework relies, in the same way, on a profiling step for building a regression model and for the prediction of the execution times of the kernels. MKMD schedules the kernels considering predicted execution times, dependencies between kernels and buffer transfer cost using a two phased approach : it first performs a coarse-grain scheduling of indivisible kernels; it then performs kernel partitioning at work-group granularity to offload work-groups of selected kernels to idle devices so as to improve previous coarse-grain scheduling results. MKMD executes kernels and performs data transfer according to the scheduling decision made prior actual execution. MKMD can therefore fail to take good scheduling decisions for irregular applications where execution time may be difficult to model and predict. Moreover, this method assume that there is enough parallelism in the task graph and is not suited for chains of kernels.

The benefit of static methods over dynamic methods to balance the workload of data parallel kernels partitioned heterogeneous multi-device architectures is that they generally have less run-time overhead as they rely on offline profiling or training. However, these methods are limited to

applications regular workloads.

4.3.2 Dynamic Coarse-grain Partitioning

In [66], the authors propose a dynamic load-balancing algorithm for a single OpenCL kernel. Chunks of contiguous work-groups with increasing size are executed on different devices. When all devices have completed a certain amount of chunks, the remaining work is partitioned across the devices using the execution time of the initial work. Their approach does not require offline training and can respond to performance variability among devices. It is limited, nevertheless, to kernels whose relative performance for the small, initial chunks of work-groups may lead to a good prediction of performance for larger chunks. Besides there is no automatic support for kernel splitting.

In [56], the authors propose an OpenCL runtime that takes a single device kernel and executes it on CPU and GPU. Load balancing is managed at runtime. While their approach dynamically balance work between one CPU and one GPU, it cannot be easily generalized for any number of devices. Besides, data is not split between sub-kernels, all arrays are transferred to all devices. Finally, the kernel transformation is achieved by hand, and not with an automatic compiler optimization.

Navarro *et. al* presents LogFit [67], an adaptive partitioning strategy in the context of parallel loops in applications with irregular data accesses. The parallel loops is split into chunks and their method dynamically finds the chunk size that gets near optimal performance for the GPU at any point of the execution, while balancing the workload among the GPU and the CPU cores. This method is limited to architectures with an integrated GPU.

Maat [43] provides a set of load-balancing methods (static, dynamic and HGuided methods). All methods, except the static one, are based on a master thread on the host in charge of assigning data-sets (packages) to the different devices. The HGuided algorithm [68] reduces the size of the packages as the execution progresses and adapts this size to the computing powers of the devices, given as parameters. While this load balancing method can handle kernels with irregular workload and dynamic load variations, it only split the data for kernel with regular access patterns. Moreover this method does not take into account the communications times induced by the dependences between kernels when partitioning a kernel sequence.

The load balancing method proposed in STEPOCL [37] to partition OpenCL kernels on multiple heterogeneous devices is similar to our Adaptive w/o Comm method. They first execute the kernel with a Uniform partitioning. Then, based on sub-kernels execution times, the partitioning is refined and the kernel is executed with this new partitioning. This process is repeated until the partitioning converges. However, contrary to our Adaptive w/o Comm method the profiling phase is performed offline. Hence, this method can handle kernels with irregular workloads, however it cannot adapt to dynamic load variations. Moreover they do not consider the data dependences between kernels.

The load balancing method proposed in ADITHE [69] specifically targets iterative applications. They first partition the workload uniformly for the first iterations of the applications. Then, the partitioning of remaining iterations is determined using the execution times of the first iterations. This method is extended in [70] for improving not only the performance but also the energy efficiency of iterative computations on integrated GPU-CPU systems. However, while this method can adapt to performance variability among heterogeneous devices it cannot cope with irregular workload.

4.3.3 Fine-grained Partitioning

Several works propose *fine-grained partitioning* methods reshaping the workload of irregular kernels to balance the load among heterogeneous devices. The key idea behind *fine-grained partitioning* is that threads on a GPU execute in a lockstep fashion: threads in the same SIMD (Single Instruction Multiple Data) group always execute the same instruction. Hence, even if the load is balanced between the devices, resources may still be underutilized if the load is imbalanced among threads from the same SIMD group.

In [71–73], Shen *et al.* present a method targeting systems comprising CPU and GPU devices with discrete memory. The input data of threads is sorted in order of their computational demand. Threads with higher computational load are assigned to the CPU. The partitioning of the NDRange is based on a performance model that considers the performance and the data copy overhead of the CPU and the GPU memory. Based on this model and after profiling, a predictor determines the optimal partitioning between CPUs and GPUs. The required prior offline analysis, however, limits the practicality of this approach for iterative applications with dynamic load variations.

The compilation and runtime system FinePar [44] relies on fine-grain partitioning and uses a sophisticated performance modeling approach using linear regression and taking both architectural differences between the CPU and GPU and data irregularity (sparse matrix, graph processing applications) in consideration. Then before actual execution, an auto-tuner determines the partitioning threshold according to the input features and selected performance model. If the computational load of a thread is higher than the threshold value, the thread is assigned to the CPU and vice-versa. A disadvantage of this method is that the model has to be trained offline for each application and each target platform, hence it is limited to application without dynamic load variations.

Finally, Cho *et. al* [74] propose an on-the-fly workload partitioning method for integrated CPU/GPU architectures. Their technique neither requires an offline analysis nor training with workloads or input data. For an OpenCL kernel, a source-to-source compiler dynamically creates profiling code that allows the runtime system to collect information about the computational load of the threads immediately before the kernel is launched. Based on this profile information, the workload is reshaped such that all threads with a high loop iteration count above a dynamically determined threshold are executed on the CPU while the GPU only executes threads below that threshold. To hide the overhead of threads profiling and reshaping, a single workload is divided into several jobs that are launched dynamically. While one job is being executed, threads reshaping is performed for the next job. This approach is limited nevertheless to integrated CPU/GPU architectures.

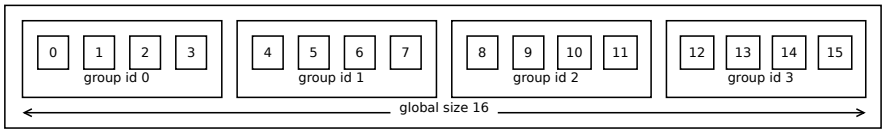
4.4 Implementation

This section provides some implementation elements of our framework. It consists in two C++ libraries: *Libsplit* and *LibKernelExpr*; and two LLVM tools: *Kernel Analyzer* and *Kernel Transformer*. The number of lines of code of each of these components is shown in **Table 4.3**.

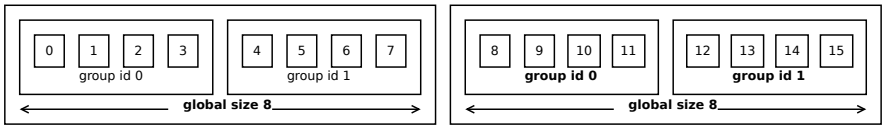
Kernel Transformation When executing a kernel with a fraction of the original NDRange, some syntactic modifications are needed in order to keep the correct semantics. Indeed, the global size is different from the original kernel, the number of work-groups has changed and their ids have changed too.

Component	LoC	Lang
Libsplit	18700	C++
LibKernelExpr	6716	C++
Kernel Analyzer	2524	C++
Kernel Transformer	579	C++

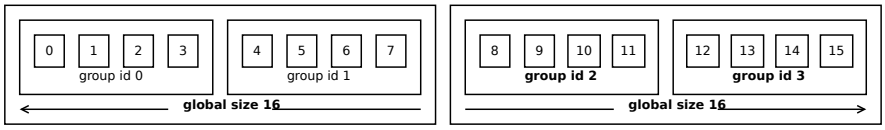
Table 4.3: Framework Components.



(a) Original NDRange.



(b) Two Sub-NDRanges before kernel transformation.



(c) Two Sub-NDRanges after kernel transformation.

Figure 4.10: Splitting a NDRange into two sub-NDRanges.

Figure 4.10a shows a NDRange of 16 threads composed of 4 work-groups. The global size is 16, the threads ids range from 0 to 15 and the work-groups ids range from 0 to 3. **Figure 4.10b** shows the two sub-NDRanges obtained after splitting this NDRange into two sub-NDRanges of same size. We can see that the two work-groups of the second sub-NDRange, corresponding to the third and fourth work-groups in the original NDRange, have group ids of 0 and 1. However, the group ids should be 2 and 3 to keep the right semantics. Furthermore, each sub-NDRange has a global size of 8 whereas it should be 16.

In order to keep the correct semantics, the *Kernel Transformer* perform a source-to-source transformation of each kernel. Two additional parameters are added to the kernels: `splitdim` (1, 2 or 3) accounts for the dimension of the NDRange that is split, and `numgroups` is the number of work-groups in this dimension. Then the following function calls are changed:

Expression	Rewritten into
`get_global_size(expr)`	`(expr == splitdim ? numgroups*get_local_size(expr):get_global_size(expr))`
`get_num_groups(expr)`	`(expr == splitdim ? numgroups:get_num_groups(expr))`
`get_group_id(expr)`	`(get_global_id(expr) / get_local_size(expr))`

After this transformation, the two sub-NDRanges keep the same semantic as the original NDRange as shown in **Figure 4.10c**. The global size remains 16 as in the original NDRange and the work-group ids the two workgroups of the second sub-NDRange are 2 and 3.

The *Kernel Transformer* is implemented as an *AST Visitor* in Clang, the frontend of LLVM [25].

LibKernelExpr The *LibKernelExpr* library implements the parametric regions presented in **Section 3.1.2**.

Kernel Analyzer The *Kernel Analyzer* is implemented inside the LLVM compiler [25] as a *function pass*. It takes as input the LLVM intermediate representation (LLVM IR) of a kernel generated with Clang and generates the parametric read and write regions of each buffer accessed by the kernel as explained in **Section 3.1**.

The *OpenCL C* language used to write compute kernels is based on the C99 but comes with several restrictions facilitating the analysis of these kernels: function pointers, bit fields and variable-length arrays are omitted and recursion is forbidden. Hence, it is possible to inline all functions called inside a kernel to avoid to resort to an inter-procedural analysis. Moreover, buffers passed as arguments to a kernel belong to a different address space from local variables and there is no heap allocation as in regular C codes. Hence, there is no need for a complex alias analysis in order to analyze an OpenCL kernel.

Our LLVM pass iterates over the load and store instructions and for each load/store instruction from/into a buffer passed as parameter of the kernel, a parametric region is build using the *LibKernelExpr* library. The LLVM IR is based on the SSA form where each variable is exactly defined one. Variables that are assigned initially in multiples statements are renamed into new instances, one per statement. Hence, the parametric regions are built by following the use/def chain.

When multiple control-flow paths join in the CFG, renamed variables are combined with a ϕ-function into a new variable instance. We only handle ϕ-function corresponding to induction variables. In order to determine the possible range of values taken by an induction variable defined by a ϕ-function we resort to the *LoopInfo* and *ScalarEvolution* passes of LLVM.

Finally, to determine the conditionals governing the execution of load and store statements, the *Iterated Post-Dominance Frontier* [45] is computed using the *PostDominatorTree* pass of LLVM.

Libsplit The *Libsplit* library is responsible for interposing calls to all the functions from the OpenCL API. In order to interpose these functions, our library is preloaded dynamically at runtime using the `LD_PRELOAD` environment variable. Hence no modification or recompilation of the original application is required in order to transform a single-device application into a multi-device application using our framework.

This library implements the *Buffer Manager* responsible for keeping track of the regions of buffers owned by each device and managing the data transfers between devices. The directories used to keep track on the data owned by each buffer are implemented as sorted lists of disjoint intervals.

The library also implements the *Dynamic Partitioner* responsible for balancing the workload of the application. We used the GNU Linear Programming Kit [75] to build and solve the linear system described in **Section 4.2.1**.

A typical OpenCL application requires calling at least the following functions from the OpenCL API. We briefly describe here how these functions are interposed by our library:

1. *clCreateContext:* This function creates a *clContext* object containing one or multiple devices from the same vendor. Hence, it is not possible to create a context containing an Intel CPU and an NVIDIA GPU for example. When using our framework, the list of devices that must be used to partition the workload are passed as command line arguments. Hence, when

this function is called a fake *clContext* object is returned containing one *clContext* and one command-queue per device selected.

2. *clCreateCommandQueue:* This function creates a command queue associated to a device and a context. The command queue can execute the command in-order or out-of-order. When this function is interposed by our library, a fake command queue is returned.

3. *clCreateProgramWithSources:* This function creates a *clProgram* object associated to a context from the source code of OpenCL kernels. When this function is interposed by our library, the source code is transformed using the *Kernel Transformer* tool. Then, one *clProgram* is created from the transformed sources for each context. Finally, a fake *clProgram* object containing the *clProgram* of each device is returned.

4. *clBuildProgram:* This function takes as parameter a device list and a *clProgram* and compiles the program on each device from the list using the vendor compiler. When this function is interposed by our library, the *clProgram* of each device is compiled.

5. *clCreateKernel:* This function takes as parameter a kernel name and a program and returns a *clKernel* object. When this function is interposed by our library and the parametric read and write regions of the kernel are built using the *Kernel Analyzer*. Then a fake *clKernel* object is returned containing one *clKernel* per device.

6. *clCreateBuffer:* This function takes as parameters a size N and a context and creates a buffer of size N associated to the context. When this function is interposed by our library, one buffer of size N is created for each device, then a fake *clBuffer* object containing the buffer of each device is returned.

7. *clEnqueueWriteBuffer:* This function allows to transfer a memory region from the host memory to a device buffer B. When this function is interposed by our library, the data transfer is deferred until kernels execution. Our library registers the commands, write protects the memory region to prevent any modification until kernels execution. Then when a kernel reading buffer B is launched and the memory region analysis in instantiated for the chosen partitioning, only the required data is transferred from the host to each device. If the host memory region is modified before kernels execution, the modification access is trapped, and the whole memory region is broadcasted to all devices before the modification occurs.

8. *clEnqueueNDRangeKernel:* This function takes as parameter a kernel, a NDRange and a command queue and enqueues a command to execute the kernel with the given NDRange.

   This function is interposed by our library and proceeds in four steps:

   (a) First, a partitioning for the kernel is determined by the *Dynamic Partitioner* and the original NDRange is split into sub-NDRanges according to this partitioning.

   (b) Second, the *Buffer Manager* instantiates the parametric read and write regions of each sub-kernel with its corresponding sub-NDRange.

   (c) Third, the *Buffer Manager* transfers for each device and for each buffer the data of the instantiated read region not already present on the device sub-buffer. Then, the

directory of each buffer read by the kernel is updated to reflect its value after these transfers.

(d) Then, a command is enqueued to the command-queue of each device to execute the kernel with its corresponding sub-NDRange.

(e) Finally, the directory of each buffer written by the kernel is updated to reflect its value after sub-kernels execution using the instantiated write-region of each sub-kernel.

9. *clEnqueueReadBuffer:* This function allows to read a region of a buffer from the device to the host. When this function is interposed by our library, the region requested is read from one or several devices depending on which devices own the elements belonging to this region.

Usage: When using our framework, the following command allows to list the available compute devices on the platform:

```
$ libsplit --list-devices

2 OpenCL platforms detected
Platform 0: NVIDIA CUDA (NVIDIA Corporation)
--- Device 0 : GPU [Tesla M2075]
--- Device 1 : GPU [Tesla M2075]
--- Device 2 : GPU [Tesla M2075]
Platform 1: Intel(R) OpenCL (Intel(R) Corporation)
--- Device 3 : CPU [Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz]
```

Then, the following command lists the available partitioning strategies:

```
$ libsplit --list-strategies

uniform
adaptivewocomm
adaptivewcomm
```

Finally, the following command automatically partition the workload of the application onto the four devices listed above using the ADAPTIVE W/ COMM strategy:

```
$ libsplit --devices="0 1 2 3 4" --strategy=adaptivewcomm
  ./myopenclapp <arg1> ... <argn>
```

4.5 Summary and conclusion of the first part

Graphic Processor Units (GPU) are ubiquitous and nowadays most computing nodes of a parallel machine consist in GPUs and multicore CPUs. To simplify the development of applications on these architectures, we presented in the previous chapter a method to automatically partition OpenCL kernels onto multiple devices. By calculating precisely the region of buffers read and written by each sub-kernel and keeping track of the region of buffers owned by each device, the method aims at minimizing the amount of data to transfer between devices.

However, once the application can be automatically partitioned, it remains to balance the workload between the devices. Load balancing is difficult to achieve in general, because the architecture is heterogeneous, the parallel application may not have a constant load and both computation and communication times have to be taken into account. We showed in this chapter that only considering the execution time of kernels onto each device is not sufficient to balance the workload of a multi-kernel application. Indeed, when partitioning two kernels and the second kernel takes as input the data produced by the first kernel, the partitioning of the first kernel may have a huge impact on the volume of data to transfer before executing the second kernel.

To tackle this issue we presented a novel automatic method to dynamically balance the workload of irregular applications with an iterated sequence of kernels onto heterogeneous devices. We showed that our partitioning method is able to handle sequences of kernels with irregular workload, dynamic load variations and takes into account the communication times between kernels induced by their respective partitioning.

To conclude, we presented in the first part of this thesis a novel method that significantly improve the programmability of heterogeneous architectures. Our method automatically transforms a single-device multi-kernel application into a portable, heterogeneous multi-device and multi-kernel application.

Part II

Detection of Collective Errors Origin in Parallel Applications

CHAPTER 5

PARCOACH Extension for a Full-Interprocedural Collectives Verification

6.1 Principle and Objective 117
6.1.1 Challenges 118
6.1.2 Principle 119
6.2 Multi-Valued Expression Detection 120
6.2.1 Enhanced SSA 120
6.2.2 PDCG: Program Data- and Control-flow Dependence Graph 121
6.2.3 Finding Collective Errors 122
6.2.4 Example 123
6.3 Related Works 124
6.3.1 Dependence Analyses Techniques 124
6.3.2 Collective Error Detection Techniques 125
6.4 Experimental Results 126
6.5 Summary and conclusion of the second part 129

The advent to exascale requires more scalable and efficient techniques to help developers to locate, analyse and correct errors in parallel applications. One major type of error that can arise in parallel programs is deadlocks. When a deadlock occurs in a parallel program, it is usually hard to identify what caused it. Some tools have been developed to help programmers in the debugging process but they often come with restrictions. The second part of this thesis investigates the automatic detection of deadlocks related to collective operations in parallel programs.

Collective operations and in particular synchronizations are widely used operations in parallel programs. They are part of languages for distributed parallelism such as MPI or PGAS (collective communications), shared-memory models like OpenMP (barriers) and languages for accelerators such as CUDA (synchronization within thread blocks, cooperative groups and at warp-level). A valid use of collective operations requires at least that their sequence is the same for all threads/processes during a parallel execution. An invalid use (*collective error*) leads to deadlocks or undefined memory state that may be difficult to reproduce and debug. Indeed, these languages do

not require that all processes reach the same textual collective statement (*textual alignment* property [76]). Hence, as soon as the control flow involving these collective operations becomes more complex, ensuring the correction of such code is error-prone.

5.1 Objectives and Principles

In the second part of this thesis we propose a new method to detect collective error origins in parallel programs. The objective of our method is to find control-flow divergences in parallel programs that may lead to the execution of different sequences of collectives by different threads or processes.

Two examples of MPI programs with potential collective errors are presented in **Figure 5.1**. In the first example shown in **Figure 5.1a**, depending on the conditional line 2, different processes may not execute the same sequence of collectives. All processes will first execute a barrier, corresponding either to the `MPI_Barrier` line 4 or to the `MPI_Barrier` line 11. But, after the execution of the first barrier some processes may call the `MPI_Broadcast` collective line 6 while others may call the `MPI_Barrier` collective line 13. This situation may lead to a deadlock at runtime. The objective of our method is to statically determine that the conditional line 2 may be responsible for a deadlock.

In the second example shown in **Figure 5.1b**, the objective of our method is to statically determine that both conditionals line 2 in `g` and line 13 in `f` may be responsible for a deadlock at runtime. Indeed, the conditional line 2 in `g` may lead to the concurrent execution of the sequence {`MPI_Barrier`,`MPI_Broadcast`} by some processes and of the function `f` by others. Then, for processes executing `f` the conditional line 13 may lead to the concurrent execution of the sequence {`MPI_Barrier`,`MPI_Broadcast`,`MPI_Barrier`} by some processes and of the sequence {`MPI_Barrier`,`MPI_Barrier`,`MPI_Barrier`} by others.

An analysis to detect conditionals leading to different sequence of collectives in parallel programs has already been proposed and implemented in the PARCOACH framework [77–80]. However, this analysis only finds control-flow divergence leading to the execution of different sequences of collectives within the same function (intraprocedural analysis).

In this chapter, we present the PARCOACH analysis and we propose to extend it to a full interprocedural analysis. A more sophisticated analysis allowing to only detect conditionals that can be evaluated differently by different processes is presented in the next chapter.

The remaining of this section describes the PARCOACH debugging method and its limitations and present the objectives of our full inter-procedural analysis.

5.1.1 PARCOACH

The PARallel COntrol flow Anomaly CHecker (PARCOACH) is a framework that detects the origin of collective errors in applications using MPI and/or OpenMP. A *collective* is defined as:

- Any blocking or non-blocking communication involving all MPI processes of a same communicator in MPI: `MPI_Barrier, MPI_Ibarrier, MPI_Bcast, MPI_Ibcast, MPI_Allreduce`, ...

```
1  void f(){
2    if(..) {
3      ...
4      MPI_Barrier(com);
5      ...
6      MPI_Broadcast(com, ..);
7      ...
8      MPI_Barrier(com);
9    } else {
10     ...
11     MPI_Barrier(com);
12     ...
13     MPI_Barrier(com);
14     ...
15     MPI_Barrier(com);
16   }
17 }
```

(a) Intraprocedural example.

```
1  void g() {
2    if (..) {
3      f();
4    } else {
5      ...
6      MPI_Barrier(com);
7      ...
8      MPI_Broadcast(com, ..);
9    }
10 }
11
12 void f() {
13   if(..) {
14     ...
15     MPI_Barrier(com);
16     ...
17     MPI_Broadcast(com, ..);
18     ...
19     MPI_Barrier(com);
20   } else {
21     ...
22     MPI_Barrier(com);
23     ...
24     MPI_Barrier(com);
25     ...
26     MPI_Barrier(com);
27   }
28 }
```

(b) Interprocedural example.

Figure 5.1: MPI examples of control-flow divergences that may lead to the execution of different sequences of collectives by different processes.

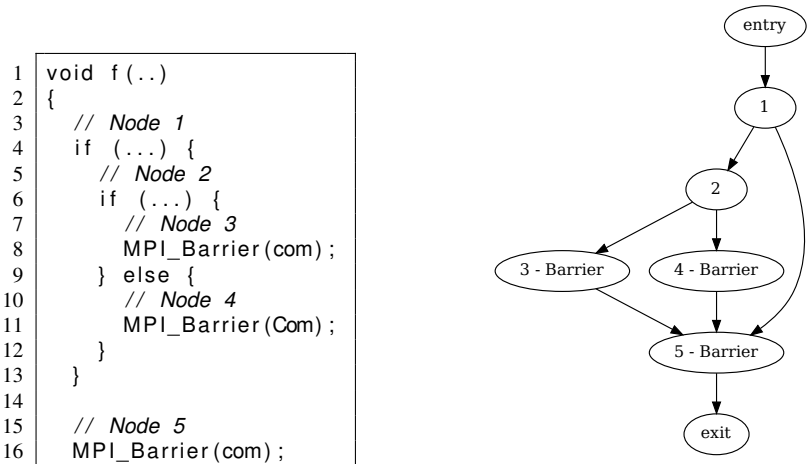
- A barrier and any worksharing construct in OpenMP:
  `#pragma omp {barrier/single/for/sections/workshare}`.
  Note that even a worksharing construct with a `nowait` *clause is considered as a collective.*

We show in this section that in certain situations PARCOACH fails to give a correct feedback to the user.

PARCOACH Analysis Principle

PARCOACH static analysis takes place in the middle of the compilation chain, where each function of a program is represented by a Control Flow Graph (CFG). In a CFG, a node can be either a straight-line code sequence called a basic block or the entry/exit point of a function and edges represent possible flow of control between nodes. For the needs of PARCOACH analysis, all CFGs are augmented with collective information: all nodes containing collectives (*collective nodes*) are tagged. For OpenMP programs, PARCOACH uses the OMPCFG representation described in [81]. The OMPCFG modifies the CFG by creating new nodes to isolate OpenMP directives and adding edges between previous nodes and the new ones according to the OpenMP semantics. Hence, nodes containing `master`, `for` and `single` directives are considered as conditionals; `sections` and `workshare` are considered as switch. An example of MPI code and its corresponding control flow graph augmented with collectives is shown in **Figure 5.2**.

PARCOACH analysis relies on the notions of *iterated postdominance frontier* [45] and *exe-*



(a) MPI code. **(b)** CFG.

Figure 5.2: Example of Control Flow Graph.

cution order to detect conditionals potentially leading to the execution of different sequences of collectives. The execution order r of a collective corresponds to its possible calling order in the program (starting with zero). When a collective has multiple possible execution orders in the program, only the maximum is kept. For example, the execution order of the barriers lines 8 and 11 in **Figure 5.2a** is 0 (first collectives encountered) and the execution order of the barrier line 16 is 1 (first or second collective encountered). The iterated postdominance frontier (PDF^+) of a node n corresponds to the control-flow divergences that may result in the execution or non-execution of n. For example in **Figure 5.2b**, $PDF^+(2) = \{1\}$. Indeed, only node 1 governs the execution or non-execution of node 2. In PARCOACH, this notion is extended to a set of nodes to detect the conditionals potentially leading to different sequences of collectives [77].

Detecting conditionals potentially leading to different sequences of collectives boils down to compute for each collective c and each execution order r the PDF+ of the set of nodes $C_{r,c}$ containing a call to collective c with execution with r. For example in **Figure 5.2b**: $PDF^+(C_{0,\text{Barrier}} = PDF^+(3,4) = \{1\}$. Indeed, depending on the conditional line 4 corresponding to node 1, the barrier with execution order 0 corresponding to the MPI barriers line 8 and 9 may not be executed by all processes.

PARCOACH intraprocedural

PARCOACH detects misuse of collectives in two steps. First, an intraprocedural static analysis studies the control flow of each function of a program to find *statically incorrect functions*: functions containing potential deadlocks [77]. During this step, warnings are issued with all conditionals potentially responsible for a deadlock. Because potential errors found at compilation-time may not be correlated with actual control-flow (false positive), the second step instruments all collectives inside *statically incorrect functions* in order to verify the potential deadlocks at execution time. Check functions are inserted before all collectives and return statements of the function. In case of an actual deadlock situation at runtime, the execution is stopped, displaying an error message with compilation information. PARCOACH aims at pinpointing the cause of collective errors

and giving the most precise feedback to developers. We use the term *intraprocedural* to refer to this analysis.

Example:
The simple MPI example shown in **Figure 5.3a** illustrates a potential issue with collective communication. Assume here that function `g` is called by all processes. Depending on the value of the input parameter `r`, a process will execute or not the barrier in the `if` statement in `f`. If `r` is not uniformly true or false among MPI processes, some tasks will be blocked in `f` while the remaining processes will reach the barrier in `g`. These processes will then terminate, while the first ones will be in a deadlock situation at the barrier in `g`.

For this example, PARCOACH *intraprocedural* emits a warning for the `MPI_Barrier` in `f` and pinpoints the conditional line 2 as a potential source of deadlock. Then the function `f` is instrumented as shown in **Figure 5.3b**. Notice that the function `g` is not instrumented by PARCOACH *intraprocedural* as this function is considered as statically correct. In order to partition processes according to their behavior regarding the conditional in `f`, two calls to the collective `MPI_Reduce` with the `equalsop` operation (bit equality checking) are inserted in the code: One before the barrier operation with the input value 1 (1^{st} parameter of the call), and one before the `return` statement with the input value 0. All processes call the `MPI_Reduce` collective, whatever their execution path. However, input values should be the same, otherwise the function is incorrect and `MPI_Abort` is issued in order to prevent from deadlocking.

```
1  void f (int r) {
2    if (r == 0)
3      MPI_Barrier(MPI_COMM_WORLD);
4    return;
5  }
6
7  void g (int r) {
8    f(r);
9    MPI_Barrier(MPI_COMM_WORLD);
10   exit(0);
11 }
```

(a) A simple example.

```
1  void f (int r) {
2    int res;
3    if (r == 0) {
4      MPI_Reduce(1,&res,1,MPI_INT,equalsop,
5                 0,MPI_COMM_WORLD);
6      if (rank == 0 && res == -1 )
7        MPI_Abort(MPI_COMM_WORLD,0);
8      MPI_Barrier(MPI_COMM_WORLD);
9    }
10   MPI_Reduce(0,&res,1,MPI_INT,equalsop,
11              0,MPI_COMM_WORLD);
12   if (rank==0 && res == -1)
13     MPI_Abort(MPI_COMM_WORLD,0);
14   return;
15 }
```

(b) The instrumented simple example.

Figure 5.3: A simple example and its instrumentation.

Limitations of the intraprocedural analysis

By analysing each function separately, the *execution order* computed by PARCOACH *intraprocedural* for each collective may be incorrect when the function analysed contains calls to functions calling collectives. Moreover, the set of nodes $C_{r,c}$ used to compute conditionals potentially leading to the execution of different sequences of collectives may be incomplete as a call to collective c with execution order r may appear in another function. Hence, PARCOACH *intraprocedural* may return *false positive* and *false negative* results.

Definition 9. A *false positive* result means that PARCOACH emits a warning for a collective *c* with execution order *r* while there is no execution path in the program where the *r*-th collective called is not *c*.

Definition 10. A *false negative* result means that PARCOACH does not emit a warning for a collective *c* with execution order *r* while there exists one execution path in the program where the *r*-th collective called is *c* and another execution path where the *r*-th collective called is not *c*.

Example:

We illustrate the limitations of the *intraprocedural* version of PARCOACH on several MPI and OpenMP examples shown in **Figure 5.4**.

For the MPI code 1, PARCOACH intraprocedural finds a potential deadlock in function `f` because of the conditional line 7 and issues a warning indicating the barrier line 8 may not be called by all processes. However, no warning is issued for the `MPI_Ibarrier` in `g`. Indeed, the intraprocedural analysis checks each function separately.

In MPI code 2, the intraprocedural analysis detects a potential deadlock in function `f` (collective sequences are `MPI_Barrier MPI_Barrier` in `then` and `MPI_Barrier MPI_Bcast MPI_Barrier` in `else`). Indeed, the intraprocedural analysis checks each function separately and does not take into account the call to `MPI_Bcast` in function `g`. This results in a false positive warning.

In MPI code 3, PARCOACH intraprocedural will report an error for the `MPI_Reduce` line 4 in `g` and the `MPI_Barrier` line 9 in `f`. However, the warning emitted for the `MPI_Barrier` line 9 is a false positive.

In MPI code 4. PARCOACH intraprocedural identifies the conditional line 2 in `g` as potentially leading to a deadlock, but not the conditional line 7 in `f`. And yet, the conditional line 7 is also responsible for a potential deadlock. The same analysis can be applied to the OpenMP code 1 (same code written in OpenMP).

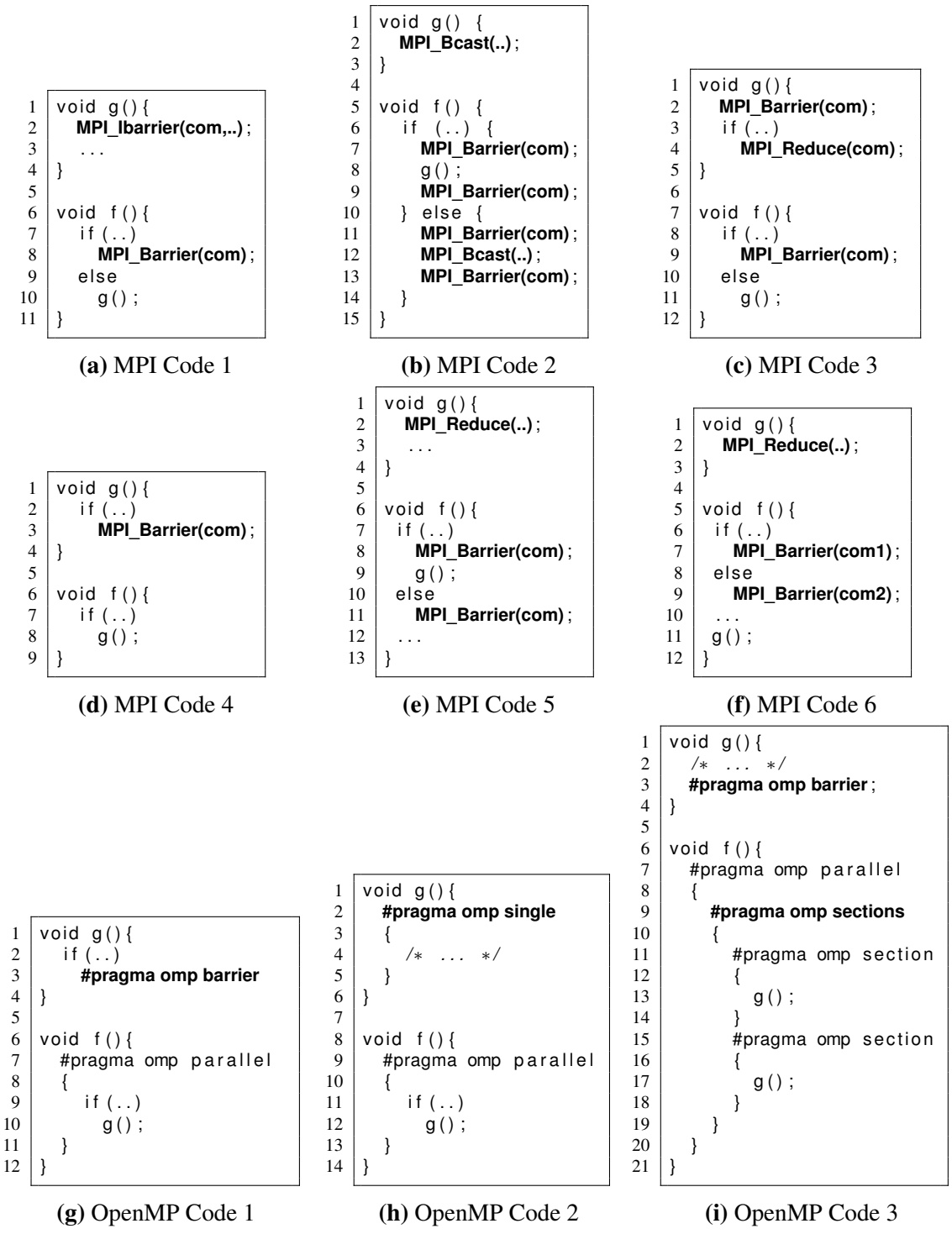
By analysing each function separately, the intraprocedural analysis doesn't detect the potential deadlock due to the `MPI_Reduce` line 2 in the MPI code 5. Both functions are considered as statically correct and none of them is instrumented. Hence, at runtime if the conditional line 7 is evaluated differently by different processes, PARCOACH will fail to prevent the deadlock. This false negative result is fixed by the summary-based analysis.

In the OpenMP code 2, the `single` line 2 may not be called by all OpenMP threads because of the conditional line 11. By analysing `g` and `f` separately, PARCOACH intraprocedural doesn't detect any collective error.

In the OpenMP code 3, the two `section` regions contain a call to `g` which contains a barrier. By default, these two regions will be executed once by two different threads. As there is an implicit barrier at the end of the `sections` construct, all threads will synchronize through distinct barriers. This is not detected by the intraprocedural analysis and may lead to a deadlock situation.

PARCOACH summary-based interprocedural

In [79], Saillard *et. al* propose a light improvement of PARCOACH static analysis to handle interprocedural information. The method keeps and reuses summaries of functions. When analysing



```
1  void g(){
2    MPI_Ibarrier(com,..);
3    ...
4  }
5
6  void f(){
7    if(..)
8      MPI_Barrier(com);
9    else
10     g();
11 }
```

(a) MPI Code 1

```
1  void g() {
2    MPI_Bcast(..);
3  }
4
5  void f() {
6    if (..) {
7      MPI_Barrier(com);
8      g();
9      MPI_Barrier(com);
10   } else {
11     MPI_Barrier(com);
12     MPI_Bcast(..);
13     MPI_Barrier(com);
14   }
15 }
```

(b) MPI Code 2

```
1  void g(){
2    MPI_Barrier(com);
3    if(..)
4      MPI_Reduce(com);
5  }
6
7  void f(){
8    if(..)
9      MPI_Barrier(com);
10   else
11     g();
12 }
```

(c) MPI Code 3

```
1  void g(){
2    if(..)
3      MPI_Barrier(com);
4  }
5
6  void f(){
7    if(..)
8      g();
9  }
```

(d) MPI Code 4

```
1  void g(){
2    MPI_Reduce(..);
3    ...
4  }
5
6  void f(){
7   if(..)
8     MPI_Barrier(com);
9     g();
10  else
11    MPI_Barrier(com);
12  ...
13 }
```

(e) MPI Code 5

```
1  void g(){
2    MPI_Reduce(..);
3  }
4
5  void f(){
6   if(..)
7     MPI_Barrier(com1);
8   else
9     MPI_Barrier(com2);
10  ...
11  g();
12 }
```

(f) MPI Code 6

```
1  void g(){
2    if(..)
3      #pragma omp barrier
4  }
5
6  void f(){
7    #pragma omp parallel
8    {
9      if(..)
10       g();
11   }
12 }
```

(g) OpenMP Code 1

```
1  void g(){
2    #pragma omp single
3    {
4      /* ... */
5    }
6  }
7
8  void f(){
9    #pragma omp parallel
10   {
11     if(..)
12       g();
13   }
14 }
```

(h) OpenMP Code 2

```
1  void g(){
2    /* ... */
3    #pragma omp barrier;
4  }
5
6  void f(){
7    #pragma omp parallel
8    {
9      #pragma omp sections
10     {
11       #pragma omp section
12       {
13         g();
14       }
15       #pragma omp section
16       {
17         g();
18       }
19     }
20   }
21 }
```

(i) OpenMP Code 3

Figure 5.4: Examples of MPI and OpenMP codes.

a function, all functions called inside this function are replaced by the valid sequence of MPI collectives they contain (collective calls not depending on the control flow). Similarly, in [78] they expose the same idea for OpenMP programs. For purpose of clarity, we use the term *summary-based interprocedural* analysis when referring to this method.

Example:

The false positive warning returned by PARCOACH *intraprocedural* for MPI code 2 is removed with the summary-based interprocedural analysis as `g` is replaced by its summary ({`MPI_Bcast`}) corresponding to the valid sequence of collectives in function `g`.

Similarly, for MPI code 3 the call to function `g` is replaced by its summary ({`MPI_Barrier`}) and the false positive warning returned by PARCOACH *intraprocedural* is removed.

In the OpenMP code 2, the call to function `g` in `f` is replaced by its summary ({`single`}), and the analysis identifies the conditional line 11 as the cause of a possible deadlock.

Limitations of the summary-based interprocedural analysis

The summary-based interprocedural analysis is limited when there is an invalid sequence of collectives in a function. In this case, the summary of the function is empty. This prevents PARCOACH from reporting correct and precise feedback as the analysis can miss the real cause of a deadlock. Moreover, the analysis does not consider the MPI communicators and can miss potential deadlocks when different processes call the same collective

Example:

In MPI code 3, the summary-based interprocedural analysis only emits a warning for the call to `MPI_Reduce` in `g`. However, as the `MPI_Reduce` does not belong to the valid sequence of collective in `g`, it does not appear during the analysis of function `f`. Hence, PARCOACH only pinpoints the conditional line 3 in `g` as a possible source of deadlock. This report is inaccurate as both conditionals in `g` and `f` (lines 3 and 8) are potentially leading to a collective error.

The same scenario is presented in MPI code 4. The summary-based interprocedural analysis identifies the conditional line 2 in `g` as potentially leading to a deadlock, but not the conditional line 7 in `f`. Indeed, there is no valid sequence of collectives. Hence, the summary of `g` is empty. And yet, the conditional line 7 is also responsible for a potential deadlock. Besides, if all processes eventually call the barrier in `g` and don't have the same value for the conditional in `f`, the feedback reported by the summary-based interprocedural analysis will be wrong. The same analysis can be applied to the OpenMP code 1 (same code written in OpenMP).

In MPI code 6, a warning should be emitted for both barriers lines 7 and 9 as `com1` and `com2` may refer to different MPI communicators. However, the PARCOACH analysis does not take the MPI communicator into account to compute the valid sequence of collective. Hence, neither the intraprocedural analysis nor the summary-based interprocedural analysis detect a potential error.

Summary

By analysing each function separately, PARCOACH *intraprocedural* analysis may return false positive (e.g. MPI code 2) or false negative (e.g. MPI Code 5) results. The summary-based analysis fix these incorrect results, however the feedback given to the user is inaccurate. When a function is statically incorrect, the summary of the function kept by the summary-based interprocedural analysis is incomplete (e.g. MPI Code 4). This prevents from reporting correct and precise feedback as the analysis can miss the real cause of a deadlock. Moreover, the analysis does not consider the MPI communicators and can miss potential deadlocks when different processes call the same collective

5.1.2 Objective of our Full-Interprocedural Analysis

The objective of our full-interprocedural analysis is to statically detect all conditionals in a parallel program potentially leading to the execution of different sequences of collectives, taking interprocedural control-flow and MPI communicators into account.

This new analysis is presented in **Section 5.2**, then a new code instrumentation is presented in **Section 5.3**.

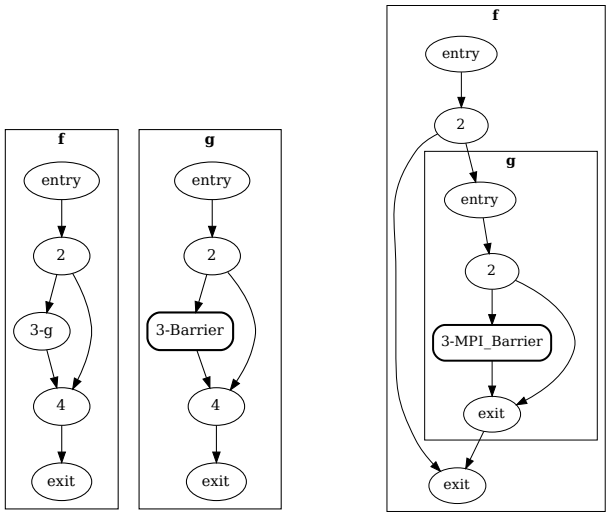
5.2 Full-Interprocedural Analysis

This section describes our new interprocedural analysis, referred as *full-interprocedural* analysis. Our method consists in building a parallel program control-flow graph capturing the control-flow of the whole program, then we rely on the principles of the PARCOACH analysis to compute conditionals potentially leading to the execution of different sequences of collectives.

5.2.1 PPCFG Construction

The full-interprocedural analysis builds a parallel program control flow graph (PPCFG) in order to get interprocedural information. We extend the intermediate representation used by PARCOACH by replacing each callsite by its CFG. In order to reduce the cost of the interprocedural analysis, each function CFG is first reduced. Only nodes with collectives, those inside the PDF^+ of these nodes, function entry and exit nodes are kept. All other nodes are removed. The edges among the nodes keep the relation of successor and predecessor existing in the initial CFG.

Figure 5.5b illustrates the PPCFG of the MPI code 4 example presented in **Figure 5.4, page 105**. The PPCFG is built based on the initial functions CFG presented in **Figure 5.5a**. Thick nodes are collective nodes, boxes represent functions.



(a) Functions CFG. **(b)** PPCFG.

Figure 5.5: MPI Code 4 functions CFG (left) and the corresponding PPCFG (right).

5.2.2 Collective Error Detection

Our analysis studies the PPCFG to find nodes conducting to paths with different sequences of collectives (i.e. not the same number or not the same collectives). With a graph traversal of the PPCFG, we compute the possible *execution order* (i.e. calling order) of each collective and the PDF^+ for collectives of the same type and order. Nodes in the PDF^+ represent all conditionals possibly responsible for a deadlock. In the MPI code 4, the barrier has an execution order 0 (first collective encountered). Conditionals lines 2 and 7 are in the PDF^+ of the instruction corresponding to `MPI_Barrier`.

Algorithm 1 describes the collective errors detection for MPI and OpenMP programs. The algorithm takes as input the *PPCFG* of a program and returns the set *O* containing the information needed to give a precise feedback to users (collective name, line of collectives and conditionals in the source code) and the set of all conditionals potentially responsible for a deadlock.

The algorithm first computes the *execution order* of each collective by computing for each node of the PPCFG the number of collectives on the execution paths from the program entry to the node. This number is 0 for nodes before the first collective (including the node with the first collective), 1 for nodes reached after one collective and so on.

When multiple paths exist, nodes can have multiple numbers. Hence, loop backedges are removed to have a finite numbering and when a collective has multiple execution orders, only the highest one is considered.

After calculating the execution order of each collective, the algorithm computes conditionals in the program leading to different sequences of collectives. These conditionals are obtained by computing for each collective c and each execution order r the PDF^+ of the set of nodes $C_{r,c}$ containing a call to collective c with execution order with r.

Whenever a function contains a collective call in a loop, it is considered as statically incorrect. The analysis does not take into account the number of iteration in the loop as this information is unknown at compile time. For MPI applications, we analyse the program separately for each communicator.

Example:

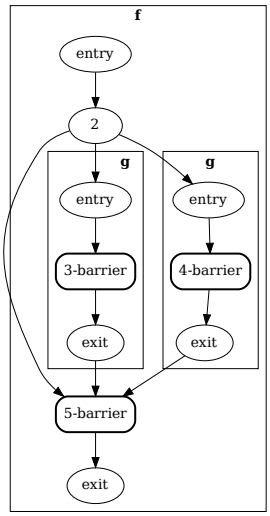
The PPCFGs of the MPI Code 1 and OpenMP Code 3 examples are shown in **Figure 5.6**. The algorithm first computes execution order of collectives. In **Figure 5.6a**, barriers in nodes 3 and 4 can be the first collectives encountered (paths $entry_f \rightarrow 2 \rightarrow entry_g \rightarrow 3$ *or* 4). The barrier node 5 can be the first collective (paths $entry_f \rightarrow 2 \rightarrow 5$) or the second collective encountered. The algorithm then considers $C_{0,barrier} = \{3, 4\}$ and $C_{1,barrier} = \{5\}$ with $PDF^+(C_{0,barrier}) = \{2\}$ and $PDF^+(C_{1,barrier}) = \emptyset$. Because $PDF^+(C_{0,barrier}) \neq \emptyset$, a warning will be issued for the conditional node 2.

The PPCFG **Figure 5.6b** has two collectives: `MPI_Barrier` in node 3 and `MPI_Ibarrier` in node 4. The static analysis detects a potential collective error for the barrier in 3 and the non-blocking barrier in 4 and reports a warning for the conditional located in node 2 ($PDF^+(C_{0,Barrier}) = \{2\}$ and $PDF^+(C_{0,Ibarrier}) = \{2\}$).

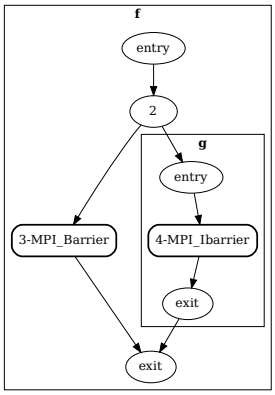
The new instrumentation step is presented in the next section.

Data: *PPCFG*
$O \leftarrow \emptyset$ ▷ Output set
Remove loop backedges in $PPCFG = (V, E)$ to compute execution orders for each collective
for *r in node orders* **do**
  for *c in collective names of execution order r* **do**
    $C_{r,c} \leftarrow \{u \in V | r$ is the max. execution order of u, u executes a collective with name $c\}$
    if $PDF^+(C_{r,c}) \neq \emptyset$ **then**
      $O \leftarrow O \cup (c, PDF^+(C_{r,c}))$
  end
  for *each collective c in a loop* **do**
    $O \leftarrow O \cup (c, \{$ loop exit nodes $\})$
  end
end
Output nodes in O as warnings
return O

Algorithm 1: Full-interprocedural Control-flow Analysis.



(a) OpenMP code 3 PPCFG.



(b) MPI Code 1 PPCFG.

Figure 5.6: MPI Code 1 and OpenMP Code 3 PPCFG.

5.3 Code Instrumentation

With the help of the static analysis, we perform a selective instrumentation of programs. Only statically incorrect programs are instrumented (no warning is issued during compilation time for statically correct programs so no instrumentation is done).

For statically incorrect programs, all collectives and exit statements are instrumented with Check Collective (`CC`) functions. The instrumentation starts from the first collectives that may

deadlock in the program. Relying on the work in [77], we define the `CC` function as follows. CC functions take as input an integer i_{model} identifying the parallel programming model used (MPI or OpenMP), the communicator related to the collective (0 in case of OpenMP), an integer i_c identifying the collective and the set O generated at compile-time. Through `CC` calls, processes/threads can verify which collectives will be called at different steps of execution. For MPI programs, `CC` calls a `MPI_Reduce` with a new MPI operator returning -1 if there is at least two different integers i_c among processes. For OpenMP programs, each thread updates a shared variable related to i_c. When a deadlock is about to occur, an error message is returned with compilation information (related warnings).

Example:
An example of MPI code instrumentation is shown in **Figure 5.7** (inspired from MPI Code 4 in **Figure 5.4d, page 105**). A CC function is inserted before the collective `MPI_Barrier` in `g` and `MPI_Finalize` in `main`. The `MPI_Barrier` line 8 is called by all processes and is therefore not instrumented.

```
1  void f() {
2    if (..)
3      MPI_Barrier(com);
4  }
5
6  int main() {
7    /* .. */
8    MPI_Barrier(com);
9
10   if (..)
11     f();
12
13   MPI_Finalize();
14 }
```

(a) MPI Code.

```
1  void g() {
2    if (..) {
3      CC(MPI, com,i_barrier,O);
4      MPI_Barrier(com);
5    }
6  }
7
8  int main() {
9    /* .. */
10   MPI_Barrier(com);
11
12   if (..)
13     g();
14
15   CC(MPI, com,0,∅);
16   MPI_Finalize();
17 }
```

(b) MPI Code instrumented.

Figure 5.7: MPI example and its instrumentation.

Lemma 1. *All deadlock situations are captured by the instrumentation and the new collectives inserted do not generate a deadlock themselves.*

Proof. We denote the sequence of collective calls executed by a process/thread in a program execution as $c_1c_2...c_n$ with c_i the i-th collective called. Our instrumentation rewrites each collective c_j into s_jc_j corresponding to the CC function with integer j and the collective c_j, starting with collectives that may deadlock. A CC function s_0 is added after all collectives (CC before exit/abort/`MPI_Finalize`). Assuming the first collective that may deadlock is the k-th collective, a sequence $c_1c_2...c_k...c_n$ then becomes $c_1c_2...s_kc_k...s_nc_ns_0$. If all collectives sequences are the same for all processes/threads, no instrumentation is done and the collectives sequences are still identical. The instrumentation does not introduce deadlocks. If a program deadlocks due to collective operations, we have the two following scenarios:

- A process/thread calls a collective c_i while another process/thread calls a collective c_j with $i \neq j$. The collectives sequences of both processes/threads only differ with their last collective and are prefixed by $c_1...c_{i-1}$. The instrumentation changes both collectives sequences

into $c_1...c_{i-1}s_i$ and $c_1...c_{i-1}s_j$. The sequences stop with s_i and s_j since $CC(-,-,i,-)$ and $CC(-,-,j,-)$ lead to an error detection and abort. The modified program no longer deadlocks.

- A process/thread calls a collective while another one exits the program. The collectives sequence of the process/thread exiting the program is $c_1...c_{i-1}$ and the process/thread calling the collective executes the same prefix sequence with one more collective c_i. The instrumentation changes both collectives sequences into $c_1...c_{i-1}s_0$ and $c_1...c_{i-1}s_i$. The sequences stop with s_0 and s_i since $CC(-,-,0,-)$ and $CC(-,-,i,-)$ lead to an error detection and abort. Again, the modified program does not deadlock.

5.4 Experimental Results

This section presents the evaluation methodology and the results obtained with our full-interprocedural analysis.

Implementation: PARCOACH was previously implemented as a GCC plugin, working with GCC version 4.7.0. The summary-based interprocedural analysis was implemented as a python script working on GCC dumped traces. We integrated both the intraprocedural and full-interprocedural analyses of PARCOACH into the LLVM [25] compiler framework, version 3.9. Our new static analysis is done at the LLVM IR level and applies **Algorithm 1** and the code instrumentation. For runtime checking, the application needs to be linked to our dynamic library (which contains `CC` function implementation).

Methodology: There exists no benchmark or application with deadlock. To evaluate the efficiency of our tool, we manually introduced errors in small codes. PARCOACH was always able to detect them. In this section, we present results we obtained on the MILC [82], Gadget-2 [83] and MPI-PHYLIP [84] applications, AMG [85] from the CORAL benchmarks, the High-Performance Linpack benchmark [86] (HPL), miniAMR and CoMD from the Mantevo project [87], IOR [88] from the NERSC benchmarks (IOR-POSIX and IOR-MPIIO), Hydro [89], and IS from the NAS benchmarks [90]. **Table 5.1** shows benchmarks and applications statistics. The second column depicts the parallel programming model used. The third and fourth columns respectively give the number of functions and collectives found in programs. The last column gives the number of communicators for MPI applications. MILC and MPI-PHYLIP are represented by the cumulative sum of all mini applications they contain. AMG is parallelized with MPI and OpenMP.

5.4.1 Static Analysis Results

The number of warnings and conditionals returned by both intraprocedural and full-interprocedural analyses is depicted in **Table 5.2** for all benchmarks and applications. We can notice that NAS-OMP IS is collective error free as no warning is emitted at compile time for this benchmark.

Application	Parallelism	# func.	# coll.	# com.
MILC*	MPI	24,242	635	253
Gadget-2	MPI	193	70	1
MPI-PHYLIP*	MPI	4,000	128	12
Bench. / mini app.	**Parallelism**	**# func.**	**# coll.**	**# com.**
Coral AMG	MPI	1,207	79	19
	OpenMP	1,207	11	-
HPL	MPI	193	3	1
miniAMR	MPI	103	43	2
IOR-POSIX	MPI	175	82	5
IOR-MPIIO	MPI	197	88	5
Hydro	MPI	99	13	1
CoMD	MPI	124	8	1
NAS-MPI IS	MPI	36	9	1
NAS-OMP IS	OpenMP	51	3	-

Table 5.1: Applications and Benchmarks Statistics.

Application	Intraprocedural		full-interprocedural	
	#warn.	#cond.	#warn.	#cond.
MILC*	114	114	498	2195
Gadget-2	21	22	68	30
MPI-PHYLIP*	65	44	65	44
Bench. / mini app.	**Intraprocedural**		**full-interprocedural**	
	#warn.	**#cond.**	**#warn.**	**#cond.**
Coral AMG	45	34	76	169
	6	2	11	48
HPL	2	1	2	1
miniAMR	20	15	32	36
IOR-POSIX	67	64	82	79
IOR-MPIIO	73	68	88	83
Hydro	11	11	13	12
CoMD	0	0	8	3
NAS-MPI IS	3	1	3	1
NAS-OMP IS	0	0	0	0

Table 5.2: Number of warnings reported and conditionals responsible for a collective error for both intraprocedural and full-interprocedural analyses.

A more detailed ratio is presented **Figures 5.8 and 5.9**. The number of conditionals added and removed with PARCOACH using the full-interprocedural method compared to the intraprocedural analysis is shown in **Figure 5.9** while the number of warnings added and removed is shown in **Figure 5.8**.

Warnings reported by the full-interprocedural analysis are mostly new warnings and few warnings were removed. The number of conditionals added and removed is also unbalanced. Adding (resp. removing) a conditional does not necessary imply adding (resp. removing) a warning since two conditionals can be responsible for the same warning.

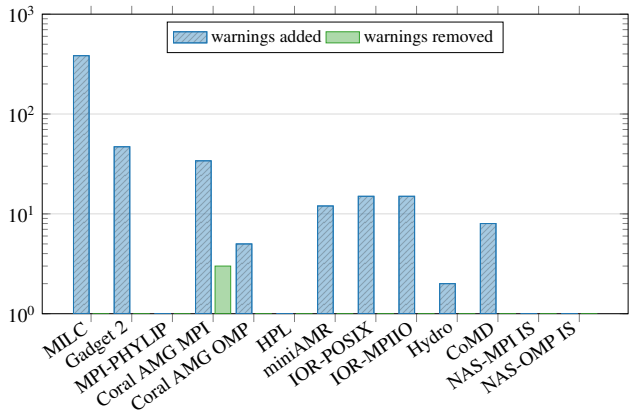


Figure 5.8: Number of warnings added and removed with PARCOACH using the full-interprocedural method compared to PARCOACH using the intraprocedural analysis.

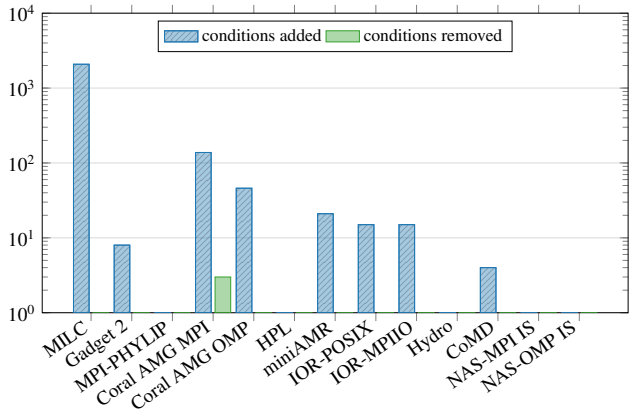


Figure 5.9: Number of conditionals added and removed with PARCOACH using the full-interprocedural method compared to PARCOACH using the intraprocedural analysis.

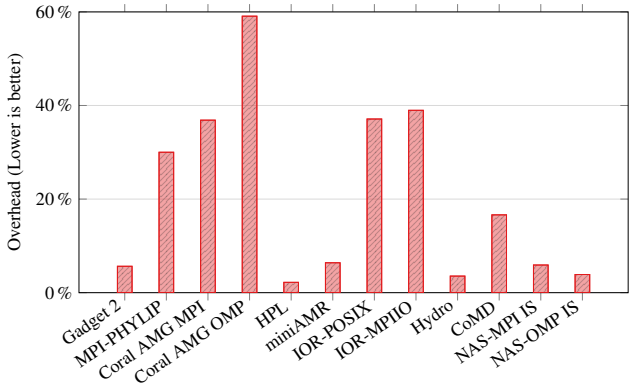


Figure 5.10: Compile-time overhead using the full-interprocedural analysis (ratio between the PARCOACH analysis time and the total compilation time).

The compile-time overhead induced when using the full-interprocedural analysis in PARCOACH is shown in **Figure 5.10**. The compilation time can be around third time the initial time (Coral AMG OMP). However, as the corresponding total compilation time with our analysis is 1 minute, we think it is acceptable.

When reporting a potential collective error, PARCOACH pinpoints the source of the error. As an example, it reports the following warning for the MPI code in **Figure 5.4a, page 105**:

```
PARCOACH: warning: MPI_Ibarrier line 2 possibly not called by all processes
because of conditional(s) line(s) 7
PARCOACH: warning: MPI_Barrier line 8 possibly not called by all processes
because of conditional(s) line(s) 7
```

This warning helps the user to prevent possible deadlocks. By looking at the conditional line 7 in the code, the developer of the application can check if this conditional is uniformly true or false for all processes. If it is the case, the warning returned is a false positive. Otherwise, this conditional may lead to a deadlock situation.

5.4.2 Execution Results

In order to realize the usability of our tool, we tested our code instrumentation on the Hydro benchmark. Hydro solves compressible Euler equations of hydrodynamics. We use the fine grain MPI version using C of the benchmark. Results were obtained on the Cori (Cray-XC40) supercomputer, deployed at NERSC [91] and averaged (over 50 runs for Hydro). Cori is composed of two partitions. One has 2,388 Intel Xeon "Haswell" nodes with 32 cores each and the other contains 9,688 Intel Xeon Phi (KNL) nodes. In this section, *Reference* denotes the original version of a benchmark.

Figure 5.11 shows the execution-time of Hydro for a range of MPI processes from 32 to 320. As can be seen in the figure, the overhead induced by PARCOACH runtime verification is under 6%.

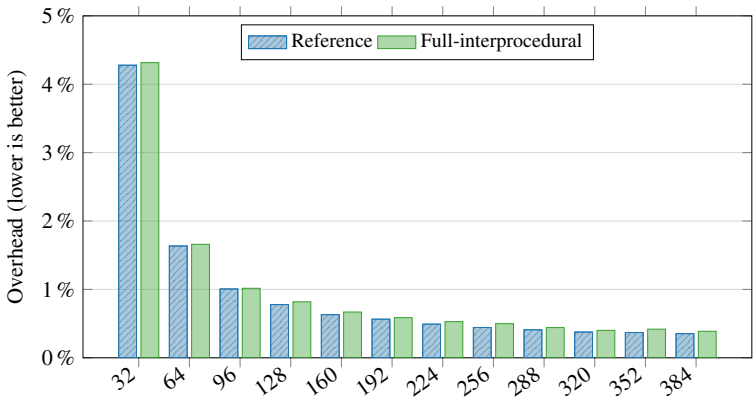


Figure 5.11: Execution-Time of Hydro with and without runtime verification (domain size = 500x500, nstepmax=200).

When a deadlock is about to occur, PARCOACH stops the execution and reports an error message with compilation information. For the code presented **Figure 5.4a**, PARCOACH reports the following error message:

```
PARCOACH: Error detected on rank 0
Abort is invoking line 8 before calling MPI_Barrier in MPIcode1.c
See warning(s):
MPI_Barrier line 8 possibly not called by all processes because of
conditional(s) line(s) 7,
MPI_Ibarrier line 2 possibly not called by all processes because of
conditional(s) line(s) 7
```

This feedback helps to fix the deadlock. The non-blocking barrier can be replaced by a blocking one.

5.5 Summary

Modern supercomputers are more and more complex to program. Consequently, optimizing parallel codes to leverage the performance of parallel architectures and at the same time keeping the readability of the code in order to prevent bugs is generally not possible. Hence, one major challenge to achieve exascale is to help developers to locate, analyse and prevent errors in parallel applications.

The second part of this thesis aims at automatically detecting collective errors origin in parallel applications. This chapter introduces the PARCOACH framework on which our contribution is based and its limitations. PARCOACH detects misuses of collectives by combining static and dynamic analyses. The static analysis detects collectives that may not be called by all processes and issues a warning for all conditionals potentially responsible for a deadlock. Then collectives are instrumented in order to verify potential deadlocks at runtime.

We have shown that in certain situations the feedback reported by PARCOACH is inaccurate and it can miss the real cause of a deadlock. To overcome these limitations we propose an extension of the PARCOACH framework. Our extension uses a parallel program control-flow graph for a more precise and accurate interprocedural analysis. This analysis brings an acceptable overhead that is not far from the overhead induced by the initial PARCOACH analysis. Furthermore, we have shown that our runtime verification has a low overhead (less than 6%) on the Hydro benchmark.

The next chapter proposes to combine the PARCOACH debugging method with a data-flow analysis in order to compute more precisely concurrent execution paths in parallel programs and reduce the number of false positive warnings.

Chapter 6

Multi-Valued Expression Analysis for Collective Checking

7.1 Automatic Tasks Adaptation for Heterogeneous Architectures 131

7.1.1 Contributions Summary 131

7.1.2 Perspectives 132

7.2 Detection of Collective Errors Origins in Parallel Programs 134

7.2.1 Contributions Summary 134

7.2.2 Perspectives 134

In the previous chapter, we have presented a method to statically detect conditionals that may lead to the execution of different sequences of collectives in parallel programs. The warnings issued by PARCOACH help developers to prevent possible deadlocks in their application.

However, different sequences of collectives must be executed concurrently for a deadlock to actually happen. If a conditional returned by PARCOACH as potentially leading to different sequences of collectives is evaluated to the same value by all processes, then all processes will execute the same branch of the conditional and therefore the same sequence of collectives. The warning issued for this conditional is therefore a false positive. Finding which collective matches a given collective therefore requires to analyze the different concurrent execution paths of a parallel execution.

Aiken and Gay first introduced the concept of *structural correctness* for synchronizations in SPMD programs, based on the notion of *multi-valued* and *single-valued* variables [92]. A variable is multi-valued if its value is dependent on the process identifier (single-valued otherwise). In structurally correct programs, processes have the same sequence of synchronization operations. If a synchronization is executed conditionally, the condition expression is single-valued. In a parallel program, determining the divergence points leading to concurrent execution of different sequences of collectives and deadlock situations corresponds to find multi-valued expression.

6.1 Principle and Objective

In this chapter, we propose to combine the inter-procedural analysis presented in the previous chapter to perform collective matching with a data-flow analysis to detect multi-valued variables.

The objective is to improve the preciseness of the PARCOACH debugging method and to reduce the number of false positives.

6.1.1 Challenges

We illustrate the challenges of detecting collective errors in parallel programs with four codes containing collectives operations in MPI, OpenMP, CUDA, and UPC.

The code of **Figure 6.1a** is written in OpenMP. According to the OpenMP specification, the same explicit and implicit[1] barriers (syntactically) should be executed by all threads. In practice, the OpenMP runtimes allow the execution of syntactically different barriers, provided all threads execute the same number of barriers. The `#pragma omp parallel` directive in function `f` defines `r`, containing the thread ID (line 10) and `s`, as a private variable. The barrier line 3 is either executed by all threads of the program or by none of them as `s` is single-valued when entering function `g` line 12. The barrier line 15 is then conditionally executed, depending on `s`. Variable `s` becomes multi-valued line 13 and leads to a deadlock situation, when the thread count is greater than 1. The objective of the analysis is to only detect the barrier line 15 as potentially dangerous. It implies being able to differentiate the two instances of variables `s` lines 11 and 13 and only detect the instance line 13 as multi-valued. Moreover, the analysis must detect interprocedural dependences between variables to check whether `s` is multi-valued in function `g`.

The MPI code in **Figure 6.1b** contains two collectives (`MPI_Barrier` and `MPI_Reduce`) and two functions: `g` and `f`. The call to `MPI_Barrier` line 17 is performed by all processes, whereas the call `MPI_Reduce` in `g` line 3 causes a deadlock. The deadlock is due to the fact that there is a multi-valued control-flow divergence line 14: Odd-ranked processes evaluate the condition to true, potentially executing the collective, while even-ranked ones evaluate it to false, hanging in the `MPI_Barrier` line 17. On the contrary, the flow-divergence line 2 does not depend on the rank, since for the unique execution path leading to line 2, `s` is single-valued. The objective of the analysis is to statically report this situation to the user, identifying the conditional line 14, and only this one, as a potential cause for mismatched calls. The analysis must not only take into account data-flow dependences but also control-flow dependences in order to detect that variable `n` is multi-valued line 14.

The CUDA code in **Figure 6.1c** manipulates multidimensional thread IDs, through predefined variables such as `threadIdx`. Synchronizations are valid if executed by all threads within the same block. Before the first synchronization in the code, the array `tile` depends on thread ID. As it is shared among threads, they all share the same version after the synchronization line 7. The synchronization line 9 is conditionally executed depending on `tile[0]`. As this value does not depend on thread ID, there is no deadlock. The third synchronization, line 11, corresponds to an invalid situation. Depending on the driver, it may lead to either a deadlock or an undefined memory configuration. This bug can be difficult to detect for a programmer in a real code, as this is a silent synchronization error. The goal of the analysis is to detect that only the synchronization line 11 may cause an error, due to the conditional line 10.

The code **Figure 6.1d** is written in Unified Parallel C (UPC) where the predefined variable `MYTHREAD` specifies thread index. In this code, because of the multi-valued expression line 3, threads with odd ID will call 9 barriers (line 5) while the others will call 10 barriers (line 9). This code is not verifiable at compile-time, hence a dynamic analysis should be proposed to prevent a

[1] There is an implicit barrier at the end of all worksharing constructs, unless a `nowait` clause is specified.

```
1  void g(int s) {
2    if(s==10)
3      #pragma omp barrier
4  }
5
6  void f() {
7    int r; int s;
8    #pragma omp parallel private(r,s)
9    {
10     r=omp_get_thread_num();
11     s=omp_get_num_threads();
12     g(s);
13     s=r%2;
14     if (s==10)
15       #pragma omp barrier
16   }
17 }
```

(a) OpenMP example.

```
1  void g(int s) {
2   if(s > 256)
3     MPI_Reduce(com, ...);
4  }
5
6  void f() {
7   int s,r,n;
8   MPI_Comm_size(com,&s);
9   MPI_Comm_rank(com,&r);
10  if (r % 2)
11    n = 1;
12  else
13    n = 2;
14  if (n == 1)
15    g(s);
16
17  MPI_Barrier(com);
18 }
```

(b) MPI example.

```
1  void f(int *data) {
2    __shared__ int tile[];
3    int tid = threadIdx.x;
4    int gid = blockIdx.x*blockDim.x+tid;
5
6    tile[tid] = data[gid];
7    __syncthreads();
8    if (tile[0])
9      __syncthreads();
10   if (tid)
11     __syncthreads();
12 }
```

(c) CUDA example.

```
1  void f() {
2   int i=1; j=10;
3   if(MYTHREAD%2){
4     while(i<10){
5       upc_barrier; i++;
6     }
7   }else{
8     while(j<20){
9       upc_barrier; j++;
10    }
11  }
12 }
```

(d) UPC example.

Figure 6.1: Examples of collective issues.

possible deadlock at runtime.

6.1.2 Principle

The principle of our analysis consists in three steps. We first computes a graph capturing data-flow and control-flow dependences between variables of the program. Then, we flood the graph from functions or variables returning thread ID/process rank in order to detect multi-valued variables in the program. Finally, we combine the PARCOACH full-interprocedural analysis with the rank-dependence information in order to only issue a warning for collective whose execution depends on multi-valued conditionals.

The static analysis we propose only detects situations and causes of *possible* deadlocks. Nonetheless, if the codes are executed with only one process/thread, there is no deadlock. Also, the following code:

```
if(A){ barrier }
if(not A){ barrier }
```

with `A` multi-valued is considered as potentially dangerous by our analysis whereas it is correct (correct but structurally incorrect). In many cases, the static analysis is able to prove that there is no deadlock situation. But when it is not the case, we can also resort to the dynamic analysis

presented in the previous chapter (cf **Section 5.3**). The dynamic analysis checks before executing collectives that indeed all processes are about to execute the same collective. The static analysis we propose helps to reduce the cost of these checks, by proving that some collectives are "safe".

6.2 Multi-Valued Expression Detection

This section presents a static analysis to find all conditionals in a program that depend on multi-valued expressions and that lead to different sequences of collectives. Two kinds of dependences can occur: data-flow and control-flow dependences. Both must be captured. In order to find multi-valued expressions, we build a *program data- and control-flow dependence graph* that highlights the rank dependence contamination in a program.

6.2.1 Enhanced SSA

Our analysis is based on the Static Single Assignment (SSA) form of the program. In SSA variables are defined by exactly one statement in the program text. Variables that are assigned initially in multiple statements are renamed into new instances, one per statement. Hence, the data-flow dependences inside a function can be captured by following def/use chains. When multiple control-flow paths join in the CFG, renamed variables are combined with a ϕ-function into a new variable instance. To capture control-flow dependences we compute an enhanced SSA where ϕ-functions are augmented with their predicates: $\phi(y_1, ..., y_k)$ is transformed into $\phi(y_1, ..., y_k, p_1, ..., p_k)$ with p_i the conditionals responsible for the choice of the y_i. These conditionals are determined by computing the PDF$^+$ of each argument y_i of the ϕ-function as in [93].

For C-like programs, variables that can be referenced with their address (address-taken variables), are only manipulated through pointers with load and store instructions in the SSA form. To compute def/use chains for address-taken variables, we rely on the principles exposed in flow-sensitive pointer analyses such as [94,95]: First a points-to analysis is computed to handle potential aliases among arrays and pointers. Then, each load $q = *p$ is annotated with a function $\mu(o)$ for each variable o that may be pointed-to by p to represent a potential use of o at the load instruction. Likewise, each store $*p = q$ is annotated with $o = \chi(o)$ for each variable o that may be pointed-to by p to represent a potential def of o at the store instruction.

There is a special case to consider for shared variables. After a synchronization (`#pragma omp barrier` in OpenMP, `syncthreads` in CUDA), shared variables have the same value for all threads. To create a new SSA instance that no longer depends on the value preceding the barrier, synchronizations are annotated with $o = \chi()$ for all shared variables o.

Then a context-sensitive *Mod-Ref Analysis* is performed to capture inter-procedural uses and def as described in [96]. The objective of this analysis is to capture the variables referenced and/or modified in functions through pointers. Each callsite cs is annotated with $\mu(o)$ for each variable o indirectly referenced in the called function. Similarly, each callsite is annotated with $o = \chi(o)$ to generate a new instance of o for each variable indirectly modified in the called function. For each address-taken variable referenced or modified in a function, a χ function is inserted at the beginning of the entry node of the CFG and a μ function is inserted at the end of the exit node of the CFG to represent their initial and final values.

Finally, all address-taken variables are converted to SSA form. This results into an augmented SSA, with value and control dependences and with additional statements in SSA describing the

effects of pointer manipulations. All possible def-use chains are then built inside SSA notation. This simplifies the construction of the following dependence graph.

6.2.2 PDCG: Program Data- and Control-flow Dependence Graph

A *program data- and control-flow dependence graph* (PDCG) is built from the enhanced SSA by connecting the definition of each variable with its uses, following the rules in **Table 6.1**. The role of this graph is to find all variables/expressions that are multi-valued. The principle is to identify the source statements that generate different values for different thread IDs/ranks, and then to propagate this property following the edges of the PDCG. The first four rules are based on [94] using similar notations. Our differences are highlighted in red.

Rule	Statement (SSA)	Edges
Value Flow Dependence		
OP	$\ell : z = x@\ell' \text{ op } y@\ell''$	$z@\ell \hookleftarrow x@\ell' \quad z@\ell \hookleftarrow y@\ell''$
PHI	$\ell : v_3 = \phi(v_1@\ell_1, v_2@\ell_2, ..., \mathbf{p@\ell_3}, ...)$	$v_3@\ell \hookleftarrow v_1@\ell_1 \quad v_3@\ell \hookleftarrow v_2@\ell_2$ $\mathbf{v_3@\ell \hookleftarrow p@\ell_3}$
LOAD	$\ell : q = *p@\ell' \ [\mu(o@\ell'')]$	$q@\ell \hookleftarrow o@\ell'' \quad \mathbf{q@\ell \hookleftarrow p@\ell'}$
STORE	$\ell : *p@\ell_1 = q@\ell_2 \ [o_2 = \chi(o_1@\ell_3)]$	$o_2@\ell \hookleftarrow q@\ell_2$ ~~$\mathbf{o_2@\ell \hookleftarrow o1@\ell_3}$~~ $\quad \mathbf{o_2@\ell \hookleftarrow p@\ell_1}$
CALL	$\ell_{cs} : r = f(..., p@\ell_1, ...) \ [\mu(o_1@\ell_2)]$ $[o_2 = \chi(o_1)]$	$q@\ell_3 \hookleftarrow p@\ell_1 \ r@\ell_{cs} \hookleftarrow x@\ell_6$

	$f(..., q@\ell_3, ...)$ {
	$[o_3@\ell_4 = \chi()]$ ... $[\mu(o_4@\ell_5)]$ *return* $x@\ell_6$
	}
		$o_3@\ell_4 \hookleftarrow o_1@\ell_2$
		$o_2@\ell_{cs} \hookleftarrow o_4@\ell_5$

Optimization		
PHI ELIM	$\ell : *p@\ell_1 = q@\ell_2 \ [o_2 = \chi(o_1@\ell_3)]$	$o_4@\ell'' = \text{fuse}(o_2@\ell, o_3@\ell', o_4@\ell'')$ $\text{remove}(o_4@\ell'' \hookleftarrow pred@\ell_4)$
	$\ell' : *p@\ell_1 = q@\ell_2 \ [o_3 = \chi(o_1@\ell_3)]$	
	$\ell'' : o_4 = \phi(o_2@\ell, o_3@\ell', pred@\ell_4)$	
RESET	$\ell_{cs} : \text{reset}(\text{buf}@\ell_1, ...) \ [\mu(o_1@\ell_2)]$ $[o_2 = \chi(o_1)]$	$\text{remove}(o_2@\ell_{cs} \hookleftarrow o_4@\ell_5)$

	reset(buf$@\ell_3$, ...) {
	$[o_3@\ell_4 = \chi()]$ ... $[\mu(o_4@\ell_5)]$
	}	

Collective Checking		
COND	$\ell : coll(...)$ with *coll* a collective of execution order r	$coll@\ell \hookleftarrow cond@\ell'$
	For all BB $\in \text{PDF}^+(C_{r,coll})$ matching: ... br cond$@\ell'$, label1, label2	

Table 6.1: Building Rules for Instructions in the PDCG. Value Flow Dependence rules are based on SVF [94] with our differences highlighted in red. Optimization rules eliminate spurious dependences and Collective Checking rule connects collectives to the conditionals governing their execution.

The `OP` and `PHI` rules correspond to straightforward data- and control-flow dependences: For an operation $\ell : z = x$ op y, the definitions of x and y at ℓ' and ℓ'' are connected to the definition of z at ℓ. For a ϕ statement $\ell : v_3 = \phi(v_1, v_2, ..., p_i, ..)$ at a control-flow join point, the definitions of the old SSA instances v_1 and v_2 at ℓ_1 and ℓ_2 are connected to the definition of the new SSA instance

v_3 at ℓ. For each predicate p_i, the definition of p_i at ℓ_3 is connected to the definition of v_3 at ℓ to handle the control-flow dependence.

The `LOAD` and `STORE` rules take into account alias information for load and store statements. For a load statement $\ell : q = *p$, the definition of each object o at ℓ'' pointed to by p is connected to the definition of q at ℓ. We also add a link from the definition of p at ℓ' to the definition of q at ℓ to denote the dependence of q with the array index. Indeed, this correspond to the case where $*p$ is $A[e]$ with e an expression. If e depends on the rank/thread ID, then q is multi-valued. Similarly, for each store instruction $\ell : *p = q$ annotated with $[o_2 = \chi(o_1)]$, the definitions of q and p are connected to o_2. However, we do not connect o_1 to o_2 since we assume that the old value o_1 is overwritten with o_2 (strong update).

The `CALL` rule handles inter-procedural dependences: For each callsite $\ell_{cs} : r = f(..., p, ...)$, the definition of the effective parameter p inside the calling function is connected to the formal parameter q in f. And the definition of the return value x in f is connected to the definition of r at ℓ_{cs}. To handle indirect value-flows for address-taken variables, given a callsite annotated with $[\mu(o_1)]$ $[o_2 = \chi(o_1)]$, the definition of o_1 in the calling function is connected to the first definition of o in f (o_3). Similarly, the last definition of o in f (o_4), is connected to the definition of o_2 at ℓ_{cs}.

The rules proposed in the Optimization section of the table correspond to two edge removal optimizations. After augmenting ϕ-nodes with their predicates, false control dependences can appear if every operand of a ϕ-node denotes the same value. This occurs in particular when considering two identical function calls in two branches of an `if..then..else` construct. Even if these two calls use the same single-valued parameters, the returned value will still depend on the predicate of the conditional (augmented SSA). To tackle this issue, the `PHIELIM` rule fuses such ϕ-nodes with their operands and disconnects the predicates. In distributed memory, after a value-sharing collective such as an all-to-all collective, the communicated buffer has the same value for all processes. This implies that this buffer does not depend on the rank after such collective, whatever its rank-dependence before the collective. To handle this situation, the `RESET` rule disconnects the path from the old SSA instance of the buffer to its new SSA instance after a value-sharing collective ($o_1@\ell_2 \hookrightarrow o_3@\ell_4 \hookrightarrow o_4@\ell_5 \hookrightarrow o_2@\ell_{cs}$). The same rule applies to any value-sharing collective where all processes receive the same result such as `MPI_Allreduce` or `MPI_Broadcast`.

Finally, to detect collectives that may not be executed by all processes, we rely on the PARCOACH full-interprocedural analysis presented in the previous chapter (cf **Section 5.2.2**). The `COND` rule connects each collective *coll* with execution order r to the conditionals contained in $PDF^+(C_{r,coll})$.

6.2.3 Finding Collective Errors

The initial method presented in **Chapter 5** finds all conditionals that may lead to the execution of different sequence of collectives. In this approach, all conditionals are assumed to be multi-valued. This is a safe over-approximation, generating many false positive results.

With our new analysis, we can now use the PDCG to identify multi-valued conditionals: We track values and nodes in the PDCG that depend on ranks, flooding the graph from *source* functions returning IDs or variables allowing tasks to identify themselves. The source functions are `MPI_Comm_rank` and `MPI_Group_rank` in MPI, `omp_get_thread_num` for OpenMP. In UPC and CUDA, the *source* is a variable: `MYTHREAD` and `threadIdx.*`.

We then use the dependence information from the PDCG to filter out false single-valued conditionals from the PDF^+ of collectives and reduce the number of false positives in PARCOACH.

After the graph flooding, only collectives whose execution depends on multi-valued conditionals are highlighted. Hence, a warning is issued only for highlighted collectives in the PDCG.

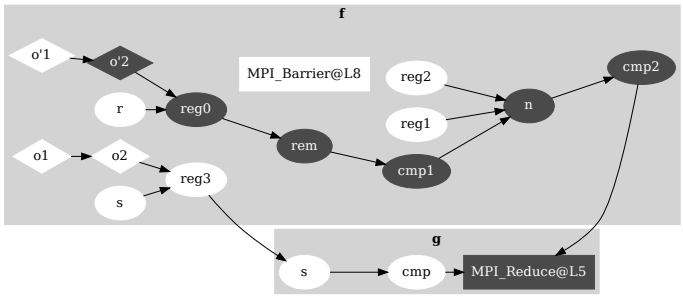
The resulting analysis still builds an over-approximation of the set of multi-valued conditionals, but a more accurate one. The augmented SSA takes into account value and control dependences, the points-to analysis provides the dependences through aliases and the inter-procedural part connect call sites and callees. Note that thanks to the PDCG, the multi-valued analysis can be path sensitive: An expression may be multi-valued or not, depending on the preceding calling context.

6.2.4 Example

Figures 6.2a and 6.2c show the enhanced SSA for the MPI code in **Figure 6.1b, page 119**. The call to `MPI_Comm_rank` is annotated with a χ function to denote the indirect definition of object `o'1` pointed-to by `r`. This generates a new SSA instance, `o'2`. Then the object `o'2` pointed-to by `r` is loaded in `reg0`. Depending on whether its value is odd or even, the execution flows to label `if.then` or `if.else`. These two control-flow paths join at label `if.end` and a ϕ-function is inserted to combine the values of `reg1` and `reg2` into variable `n`. Note that the predicate `cmp1` is added to the ϕ-function to indicate its value depends on `cmp1`. Finally, depending on whether `n` value is equal to `1` (`cmp2`) there is a conditional branching to label `if.then2` and function `g` is called. Otherwise, the function returns.

```
1  define void f() {
2    s = alloca_o; // object o1
3    r = alloca_o';// object o'1
4    MPI_Comm_size(com, s); [μ(o1)]
5    [o2 = χ(o1)]
6    MPI_Comm_rank(com, r); [μ(o'1)]
7    [o'2 = χ(o'1)]
8    reg0 = load r [μ(o'2)]
9    rem = reg0 % 2;
10   cmp1 = rem != 0;
11   br cmp1, if.then, if.else
12 if.then:
13   reg1 = 1;
14   br label if.end
15 if.else:
16   reg2 = 2;
17   br label if.end
18 if.end:
19   n = φ(reg1, reg2, cmp1)
20   cmp2 = n == 1;
21   br cmp2, if.then2, if.end2
22 if.then2:
23   reg3 = load s; [μ(o2)]
24   g(reg3);
25   br label if.end2
26 if.end2:
27   MPI_Barrier(com);
28   ret void;
29 }
```

(a) Function `f` SSA.



(b) PDCG.

```
1 define void g(i32 s) {
2   cmp = s > 256
3   br cmp, if.then, if.end
4 if.then:
5   MPI_Reduce(com, ...);
6   br if.end
7 if.end:
8   ret void
9 }
```

(c) Function `g` SSA.

Figure 6.2: Enhanced SSA form of the MPI code **Figure 6.1b** and its corresponding PDCG.

Figure 6.2b shows the corresponding PDCG. Rectangle nodes represent collectives. Diamond and circle nodes respectively represent definitions of address-taken and top-level variables (vari-

ables never referenced by their address). In this example the source `MPI_Comm_rank` is in line 6 and the first rank-dependent object is `o'2`. All library functions have mocked-up CFGs, tagging output values as rank-dependent when necessary. Out of clarity, the CFG of the MPI functions are not represented in the figure. The graph highlights the rank-dependent path from `o'2` to the call to `MPI_Reduce` in `g` passing through the conditional `cmp2` in `f`.

In this example, the execution order computed by PARCOACH for the `MPI_Reduce` line 5 is 0 (first collective encountered) while the execution order of the `MPI_Barrier` line 8 is 1. The execution of `MPI_Reduce` depends on $\text{PDF}^+(C_{0,\texttt{MPI\_Reduce}}) = \{(\texttt{cmp}, \texttt{cmp2}\}$. Indeed, the call to `MPI_Reduce` depends on the value of `cmp` in `g` and the call to `g` depends on the value of `cmp2` in `f`. Hence, `MPI_Reduce`@ℓ_5 is connected to `cmp` and `cmp2`. However, there is no path from `o'2` to `cmp`, as it does not depend on the rank.

Finally, the execution of `MPI_Barrier` is not governed by any conditional as $\text{PDF}^+(C_{1,\texttt{MPI\_Barrier}}) = \{\emptyset\}$. Hence, `MPI_Barrier`@$\ell_8$ is not connected to any node in the graph and it cannot be reached from a *source* statement (statement ℓ_7),

Since the only collective highlighted in the graph corresponds to the `MPI_Reduce` in `g` and only one of the two conditionals governing its execution is highlighted, our new analysis only issues a warning for the multi-valued conditional line 21 in `f` and the call to `MPI_Reduce` in `g`.

6.3 Related Works

This section summarizes works on dependence analyses and gives an overview of existing tools for collective error detection in parallel programs.

6.3.1 Dependence Analyses Techniques

Dependence analyses are the cornerstones of many optimizations/analyses in compilers. For instance, dependences are used for *Taint Analysis* [97–99] to determine how program inputs may affect the program execution and exploit security vulnerabilities, *Information Flow Tracking* [100–103] to prevent confidential information from leaking, static *Bug Detection* [104, 105] or code optimization and parallelization (e.g. the polyhedral model [106]). One of the difficult issues when computing data dependences is to deal with pointers/memory aliases and non scalar variables (e.g. arrays, structures). In SVF [94] the authors annotate load and store instructions with μ and χ functions to transform address-taken variables into an SSA form. However, they do not take into account the possible dependence of the pointer itself (through an array index for instance) when they build the data dependence graph.

Many of the aforementioned analyses only consider data dependences although Slowinska *et al.* [107] showed that omitting control dependences can be a huge source of false negative results. In [108], the authors introduced the concept of *Strict Control Dependences* to reduce the number of false positives in *Taint Analyses* and *Lineage Tracing*. In Parfait [109] the authors propose to extend ϕ-functions with predicates in order to handle control dependences. However address-taken variables are not transformed into SSA form.

6.3.2 Collective Error Detection Techniques

MPI Static analyses operate on the source code of an application and have the advantage of not requiring execution. They are usually based on model checking and symbolic program execution, limiting their applicability (the number of reachable states to consider is combinatorial). TASS [110] uses this approach. It builds a model by associating symbolic expressions to all input values and then checks reachable states of this model. The method presented by Zhang and Duesterwald in [76] is the closest to our work. It detects synchronization errors with an inter-procedural barrier matching technique for SPMD programs with textually unaligned barriers. Compared to our analysis, the method has no pointer analysis and all references to an array are assumed to be multi-valued. In the following section, we compare the results obtained when using the analyses proposed in [109] and [94] instead of our multi-value analysis, described in **Section 6.2**. This shows that a dedicated dependence analysis is required to preserve the correctness of the results, and the analysis we propose is more accurate than the others.

Although dynamic tools are input data set dependent and can only detect an error when it is about to occur, they better manage huge number of processes compared to static analyses. Some dynamic tools detect deadlocks with a time-out approach, which may cause false positive. It is the case of the Intel Trace Analyzer and Collector [111] (ITAC) and DAMPI [112]. Although PARCOACH static analysis may cause false positives, the program instrumentation assures only real deadlocks are catched at runtime. MUST [113, 114] is able to check MPI collective operations with an offloading approach using wait-for graphs. Compared to MUST, PARCOACH stops the execution before a deadlock occurs and gives a more precise feedback about what caused it. STAT [115] uses a post-mortem analysis. It studies the stack trace of execution to detect deadlock situations. This method does not allow finding deadlock root causes. An extension of MPICH directly verifies collective operations inside the MPI implementation [116]. This method is therefore limited to the information available in the MPI routines. PARCOACH was designed to take the best of static and dynamic methods. It combines a scalable static analysis based on the study of programs control flow with an instrumentation of the code that verifies potential deadlocks at execution-time.

OpenMP The OpenMP Analysis Toolkit (OAT) [117] relies on symbolic analysis to detect concurrency errors, including deadlocks. It encodes OpenMP regions into SMT formulas and uses the SMT-solver Yices to detect errors. Zhang *et al.* [118] detect textually unaligned barriers with an interprocedural concurrency analysis. This one uses the control flow of a program and a barrier tree. Our method is simpler: we build a parallel program control-flow graph to get interprocedural information. Compilers like GCC [119], ICC [120] or LLVM [25] issue either a warning or an error message for invalid nesting of regions. For example, GCC issues a warning for a `single` directive in another `single` directive whereas ICC and LLVM return an error message. However, if the nested region is encapsulated into a function, they don't detect anything. PARCOACH [78] uses the same static/dynamic method as for MPI programs to detect misuse of barriers and worksharing constructs in OpenMP programs. Intel Thread checker [121, 122] (now superseded by Intel Inspector XE [123]) and Sun thread analyzer [122, 124] both use code instrumentation to collect operations on memory, thread management and synchronization at runtime. These operations are recorded in a trace file which is then analyzed in order to find deadlocks. This post-mortem method has the same drawback as the dynamic methods, it only finds errors related to the parts of the program that have been executed and is dependent to the input data set.

OpenCL and CUDA For CUDA, Memcheck [125] performs checks for data races and barrier divergence. OCL grind [126] simulates OpenCL codes for debugging race conditions, runtime errors or invalid memory accesses. Both are dynamic approaches. For static verification, GPUVerify [127] checks for barrier divergence. Compared to our work, GPUVerify checks that all threads execute the same barriers syntactically without finding the conditions responsible for divergence. KLEE-CL [128] statically performs race detection, finds mismatches between C and OpenCL codes but does not focus on collective checks.

UPC UPC-SPIN [129] generates finite models of UPC programs in the modeling language of the SPIN model checker. It analyzes all possible control paths but faces a combinatorial time and memory explosion, limiting its application to small and moderate sized applications. UPC-CHECK [130] is a dynamic tool that inserts calls before and after each UPC operation and before return and exit statements in order to detect deadlocks.

6.4 Experimental Results

This section presents the evaluation methodology and the results obtained with our new analysis to detect collective errors in parallel programs.

Methodology: The analysis we propose is implemented as a link time optimization pass in the LLVM framework 3.9 integrated into PARCOACH. This section shows the results obtained on 3 HPC applications (MILC [82], Gadget [83] and MPI-PHYLIP [84]), 4 mini HPC applications (CoMD and miniAMR from the Mantevo project [87], Hydro [89] and miniGMG [131]) and 5 widely used benchmarks (HPL [86], IOR [88], AMG [85], NAS IS [90,132], and the CUDA benchmarks from Rodinia [133]). **Table 6.2** gives a short description of the applications and benchmarks we tested. The second column gives the programming language, the third column the number of lines, the fourth and fifth column respectively provide the total number of functions and collectives.

Parcoach Improvement **Figure 6.3** shows the impact of combining our multi-valued expression analysis with PARCOACH compared to the initial PARCOACH analysis. Warnings are collectives that may lead to deadlocks, and the conditionals correspond to multi-valued conditionals governing the execution of unsafe collectives. The figure displays the percentage of warnings and conditionals filtered out with our analysis compared to the initial PARCOACH analysis. The total number of filtered warnings and conditionals is given at the top of each bar. 100% for a warning bar means that the application is collective error-free (all warnings are removed, the code is safe), 0% means that our multi-valued analysis has no impact. OMP IS and miniGMG are not represented on the figure since no warning was issued for them. About half conditions are filtered out by our analysis for most applications. All warnings and conditionals are removed for Coral AMG OMP, MPI IS, UPC IS, and Rodinia.

Comparison with Related Works We compare the performance of our value-flow analysis applied to rank-dependence, w.r.t. the value-flow analyses proposed by SVF and Parfait. All three

Application	Lang	LoC	#func.	#col.	#?c
MILC*	MPI	172,290	22,242	635 (253)	566
MPI-PHYLIP*	MPI	129,821	4,000	128 (12)	65
Gadget	MPI	17,804	193	70 (1)	69
Bench./mini app.	**Lang**	**LoC**	**#func.**	**#col.**	**#?c**
Coral AMG	MPI	109,607	1,207	79 (19)	79
	OMP	109,607	1,207	11	11
HPL	MPI	34,375	193	3 (1)	3
Rodinia*	CUDA	15,956	512	219	219
CoMD	CUDA	12,920	102	39	39
miniAMR	MPI	9,364	103	43 (2)	43
IOR-MPIIO	MPI	6,501	197	88 (5)	88
Hydro	MPI	5,848	99	13 (1)	13
CoMD	MPI	5,644	124	8 (1)	8
miniGMG (AGG BAR version)	UPC	2,113	184	5	5
NAS-UPC IS	UPC	1,343	48	8	8
NAS-MPI IS	MPI	1,371	36	9 (1)	9
NAS-OMP IS	OMP	1,273	51	3	0

Table 6.2: Applications and Benchmarks Characteristics. OMP=OpenMP.

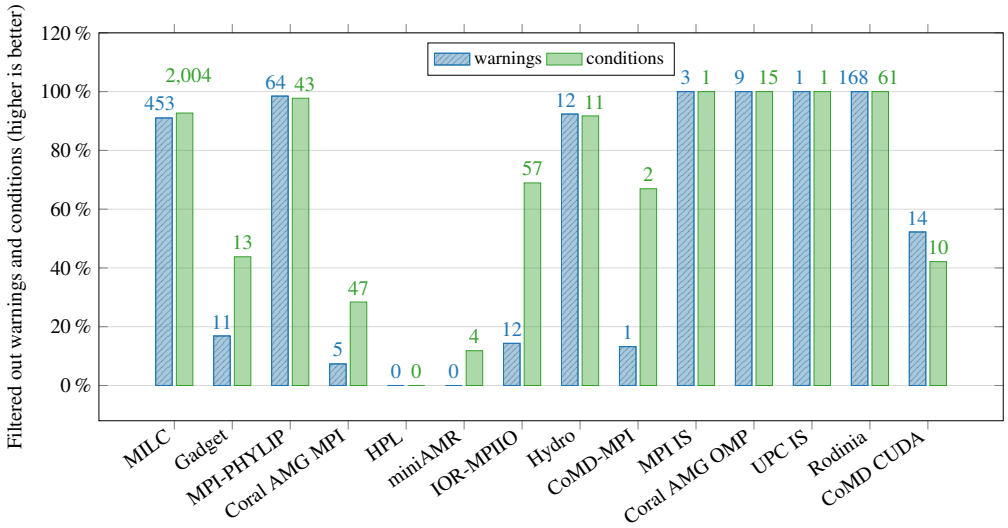


Figure 6.3: Percentage of warnings and conditionals filtered by our analysis. 100% means that the analysis has shown the program is free of collective error. The total number of filtered warnings and conditionals is given at the top of each bar.

analyses are then combined with the same inter-procedural path analysis. For the comparison, we use the 4 MPI codes presented in **Figure 6.4**, all inspired from existing benchmarks.

The first code (**Figure 6.4a**), from Hydro benchmark, defines a structure `_hydroparam` with the field `mype` that is multi-valued. For this code, all three analyses consider the entire structure as multi-valued and emit a false positive warning for the conditional line 11. Contrary to Parfait, SVF

```
1  struct _hydroparam {
2    int mype;
3    int nproc;
4  }hydroparam_t;
5
6  void f(hydroparam_t * H) {
7
8    MPI_Comm_rank(com,&H->mype);
9    MPI_Comm_size(com,&H->nproc);
10
11   if (H->nproc > 1)
12     MPI_Barrier(com);
13 }
```

(a) field-sensitive.c

```
1  void f() {
2    int r;
3    int v=0;
4    MPI_Comm_rank(com,&r);
5
6    if (!r)
7      v = 1;
8    else
9      v = 2;
10
11   if (v == 2)
12     MPI_Barrier(com);
13 }
```

(b) phi-cond.c

```
1  void f(int a) {
2    if (a > 0)
3      MPI_Barrier(com);
4
5    MPI_Comm_rank(com,&a);
6
7    if (a > 0)
8      MPI_Barrier(com);
9  }
```

(c) pointer-instance.c

```
1  void f() {
2    int r, s;
3    MPI_Comm_rank(com,&r);
4    MPI_Comm_size(com,&s);
5
6    int *A = malloc(s);
7    for (int i=0; i<s; i++)
8      A[i] = i;
9
10   if (A[r] > 10)
11     MPI_Barrier(com);
12 }
```

(d) index-dep.c

Figure 6.4: Examples of MPI codes.

and PARCOACH can handle this case by changing the pointer analysis used for a field-sensitive one.

In the code `index-dep.c`, array `A` is single-valued, however the position of the element read in the conditional line 10 depends on the rank. Hence, $A[r]$ is multi-valued. For this code only PARCOACH and Parfait issue a warning for the `MPI_Barrier` line 11.

In the code `phi-cond.c`, variable `v` is assigned in both branches of an `if..then..else` construct. Then, depending on the value of `v` after the `if..then..else`, the `MPI_Barrier` line 12 is executed. In LLVM IR, values of `v` from both branches of the conditional are combined into a ϕ-function. Detecting that `v` is multi-valued then requires to handle control-flow dependences for ϕ-functions. We added the lines involving variable `v` in miniAMR. For this example, SVF does not detect the conditional line 12 as potentially dangerous.

In `pointer-instance.c`, the parameter `a` passed to function `f` does not depend on the rank of the process. Hence, the barrier line 3 is executed by all MPI processes and the value of `a` only becomes multi-valued after the call to `MPI_Comm_rank`. In LLVM since the address of `a` is taken, it is only manipulated through a pointer with load and store operations. Making distinctions between different values of `a` then requires to compute the SSA form for address-taken variables. For this example Parfait emits a false positive warning for the `MPI_Barrier` line 3.

SVF misses dependences since it does not take into account control dependences for ϕ-instructions and index dependences for load and store instructions. Parfait does not manage SSA for pointers, ϕ-functions and control dependences for loads and stores. Besides, it can over-approximate the graph since there are backedges added to connect each instruction to its pointer operands and formal pointer arguments of callsites to their actual arguments. **Table 6.3** gives a summary of the

results. It shows that our method improves correctness w.r.t. SVF and accuracy w.r.t. Parfait. For the remaining false-positive results, a more precise dependence analysis is required. This is left for future work.

Program	PARCOACH	SVF	Parfait
field-sensitive	FP	FP	FP
index-dep	✓	FN	✓
phi-cond	✓	FN	✓
pointer-instance	✓	✓	FP

Table 6.3: Multi-valued detection comparison between PARCOACH, SVF and Parfait, both combined with collective deadlock detection. FP = false positives, FN = false negative.

The method presented can be applied incrementally: When developing a code, only the most recent code is analyzed, the rest of the code is assumed to be correct. The warnings issued then can tell the user where to check to preserve correctness in the new code.

6.5 Summary and conclusion of the second part

Collective operations and in particular synchronizations are widely used operations in parallel programs. A valid use of collective operations requires at least that their sequence is the same for all threads/processes during a parallel execution. However, all threads don't have to reach the same textual collective statement for the program to be correct. Hence, as soon as the control flow involving these collective operations becomes more complex, ensuring the correction of such code is complex. The previous chapter presented a method to automatically determine conditionals leading to different sequences of collectives in parallel programs. However, for a deadlock to happen, these conditionals must be evaluated differently by different threads. Consequently, determining the divergence points leading to concurrent execution of different sequences of collectives and possible deadlock situations corresponds to find multi-valued expression.

We presented in this chapter a new method to detect multi-valued variables in parallel programs. By combining the deadlock detection method presented in the previous chapter with a multi-valued variable detection, our new analysis was able to dramatically reduce the number of false positive results returned by PARCOACH. The analysis resorts to an inter-procedural static analysis that can prove in some cases that a program is free of collective error. The method has been applied successfully on different languages (MPI, CUDA, OpenMP, UPC) and is implemented in LLVM in the PARCOACH tool. Experiments have shown that our analysis leads to significant improvement over existing debugging method. Furthermore, through a more precise use of alias and control dependences, our static analysis outperforms existing data-flow analyses bringing additional preciseness (removing spurious dependences) and correctness (adding missing dependences).

To conclude, we presented in the second part of this thesis a novel method that significantly ease the debugging and correctness verification of complex applications with extreme scale.

Chapter 7

Conclusion and perspectives

One major concern to achieve exascale is the programmability of heterogeneous architectures. New automatic methods are required to relieve the programmer from the burden of managing the details related to the underlying architecture where its code is executed. In addition, as scientific applications are becoming more and more complex and are supposed to run at extreme scale, new tools are required to assist developers in the debugging phase of application development.

This thesis explores the combination of static and dynamic methods to improve programmability of HPC applications and is organized around two different challenges: the automatic task adaptation for heterogeneous architectures and the automatic detection or collective errors origin in parallel programs. This chapter details the contributions made and the perspectives envisaged for each of the two challenges targeted in this thesis.

7.1 Automatic Tasks Adaptation for Heterogeneous Architectures

Programming heterogeneous architectures is very difficult and error-prone. For each parallel task the programmer usually has to write as many versions as there are different architectures. Moreover, the responsibility of mapping tasks to devices, managing data transfers and balancing the load between the devices is left to the programmer. The first part of this thesis aimed at simplifying the development of applications onto heterogeneous architectures.

We rely on a programming model were the programmer expresses the parallelism of his application through a sequence of parallel loops without considering issues related to the underlying architecture where its code is executed. Then, our method automatically partitions the tasks into sub-tasks executed by each device to take full advantage of the machine capabilities. The method proposed is completely transparent to the user and automatically transforms a single device multi-kernel application into a portable, heterogeneous multi-device and multi-kernel application.

7.1.1 Contributions Summary

These contributions have been published and presented in the International European Conference on Parallel and Distributed Computing (Euro-Par) 2016 [26] and in the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2018 [27].

- The first contribution is the automatic partitioning of irregular tasks to heterogeneous architectures. Contrary to related work, the method we propose allows to automatically partition applications with complex kernels that may contain indirect memory accesses and atomic operations. By calculating precisely the region of buffers read and written by each sub-kernel and keeping track of the region of buffers owned by each device, our method minimizes the amount of data to transfer between devices.

- The second contribution is a novel automatic method to balance the workload of irregular applications with an iterated sequence of kernels onto heterogeneous devices. Our method is purely dynamic and does not require prior profiling nor sampling of the application. We showed that our load balancing method is able to handle sequences of kernels with irregular workload, dynamic load variations and takes into account the communication times between kernels induced by their respective partitioning.

7.1.2 Perspectives

While the contributions presented significantly improve the programmability of heterogeneous systems, further improvement is required. This section presents some opportunities of improvement.

(a) Load Balancing method for Non-iterative Applications

Although the partitioning method we propose can cope with any type of application, our load balancing method is limited to iterative applications. Indeed, the partitioning determined for iteration t is based on the execution time of iteration $t-1$.

Assuming the workload of each kernel of the application is large enough, a possible solution would be to split each kernel into m sub-kernels $k_1, \ldots, k_m$, thereby transforming each kernel into an iterated kernel, repeated m times. The first iteration corresponding to the execution of the first sub-kernel, the second iteration corresponding to the execution of the second sub-kernel and so on. Then each sub-kernel could be partitioned onto the different devices and the partitioning of each sub-kernel k_i could be automatically determined with our method based on the execution time of sub-kernel k_{i-1}.

However, this solution may not work for irregular kernels. Indeed, the partitioning decision for sub-kernel k_{i-1} may not be a good decision for partitioning sub-kernel k_i.

(b) Buffers Allocation and Memory Footprint

An important limitation of our method for partitioning kernels onto multiple devices is that we do not reduce the memory footprint onto each device. Hence, using our method it is not possible to take advantage of multiple devices in order to execute OpenCL kernels with large workloads that do not fit in the memory of a single device. Indeed, when a buffer is created in the original application (`clCreateBuffer`), we create a sub-buffer of the same size onto each device, even if each device will only use part of the allocated memory for its sub-buffer.

In order to reduce the memory footprint on each device, it would be necessary to know for each buffer the region accessed by each device at the time of its creation. To do so, the two following issues must be addressed:

- First, when a buffer is allocated, it is not associated with a kernel. Hence, when the user requires the creation of a buffer, it is not possible to determine which kernel requires the data allocated on this buffer.

  To overcome this limitation, a possible solution would be to combine our static analysis of computation kernels code with a static analysis of the host code in order to match each buffer with the kernels that use them as input or output.

- Second, even if for each buffer the kernels associated with it were known at the time of its creation, reducing the memory allocated onto each device would require determining for each kernel the region of the buffer accessed by each device. However, these regions can only be determined once a partitioning for these kernels has been selected. Moreover, even if the partitioning of each kernel were known in advance, for some kernels, the memory region accessed by each device does not only depend on the selected partitioning but also on values that are only known just before executing the kernel (e.g. indirection or scalar values that depend on the result of another kernel).

(c) Extension to clusters

Our method significantly improves the programmability of a single node equipped with multiple heterogeneous devices. However, as supercomputers consist in many interconnected nodes, a natural perspective of improvement would be to extend our partitioning method to a cluster.

To do so, the first requirement would be to address the memory footprint issue presented in the previous paragraph. Then our method should be adapted to manage data transfers between different nodes and our load balancing strategy should be adapted to take into account these new communication times.

(d) Integration inside a Task-Based Runtime

The parallelism model we propose to exploit the compute capabilities of multiple devices is poorly suited when there is sufficient parallelism in the task graph of the application to schedule the kernels on different devices and the workload of each kernel is not large enough to partition each kernel onto different devices. This type of application would better benefits from a task-based runtime such as StarPU [30].

With StarPU, the programmer expresses the parallelism of its application by defining tasks and dependencies between them. Then, these tasks are automatically scheduled at runtime on the available devices. However, this approach is limited when there is not enough parallelism in the task graph of the application to schedule the kernels onto different devices and keep all devices occupied. This is the case of iterated sequences of kernels like the SOTL application presented in **Section 3.2** for instance.

An interesting future work would be to integrate our partitioning method into a task-based runtime in order to combine the benefits of both scheduling and partitioning approaches. By combining both approaches, tasks granularity could be automatically adapted at runtime.

To do so, we could rely on the method proposed by Lee *et. al* [42]. In their approach, they first perform a coarse-grain scheduling of indivisible kernels, and then perform kernel partitioning at work-group granularity to offload work-groups of selected kernels to idle devices

7.2 Detection of Collective Errors Origins in Parallel Programs

With extreme scale and complexity of applications enabled by exascale systems, conventional debugging techniques are no longer appropriate and new automatic methods are required to assist developers during the debugging stage of application development.

The second part of this thesis aimed at improving the programmability of complex application for the exascale era by easing the debugging stage of their development. We proposed a novel method that automatically detects and prevents deadlocks related to collective operations in parallel applications.

7.2.1 Contributions Summary

These contributions have been presented in the International Workshop on Software Correctness for HPC Applications 2018 [28] and in the International European Conference on Parallel and Distributed Computing (Euro-Par) 2019 [29].

- The first contribution is the implementation of a full inter-procedural analysis automatically detecting conditional potentially responsible for deadlocks in parallel programs. This analysis has been implemented into the PARCOACH framework [77–80].
- The second contribution is a new static analysis that detects multi-valued expression in parallel programs. We used this analysis to filter out false positives in PARCOACH and showed that our analysis leads to significant improvement over existing debugging methods.

7.2.2 Perspectives

Naturally, further improvement is required to help application developers debugging their applications. This section presents some opportunities for improvement.

(a) User Hint

Some warnings returned by PARCOACH at compile-time are false positives as in certain situations our analysis fails to prove that a variable is single-valued.

The main reason we have identified for these false positives is that the pointer analysis that we use as input for our analysis is too conservative. For example, the alias analysis we used is field-insensitive. Hence, when a field of a structure contains a multi-valued variable, all the structure is considered as multi-valued. To overcome this limitation, the developer could annotate the conditionals of which he is certain that they are single-valued. **Figure 7.1** illustrates such an example where the structure `H` has two fields: `mype` and `nprocs` and only `mype` is multi-valued. With our analysis all the structure is considered as multi-valued, consequently the conditional line 12 depending on `nproc` is considered as multi-valued and a false positive warning is returned for the `MPI_Barrier` line 13. To overcome this limitation the user could annotate the conditional as single-valued using a pragma as shown line 11.

(b) Instrumentation Minimization

```
1  struct _hydroparam {
2    int mype;
3    int nproc;
4  }hydroparam_t;
5
6  void f(hydroparam_t * H) {
7
8    MPI_Comm_rank(com,&H->mype);
9    MPI_Comm_size(com,&H->nproc);
10
11 #pragma singlevalued
12   if (H->nproc > 1)
13     MPI_Barrier(com);
14 }
```

Figure 7.1: Annotation Example.

The instrumentation proposed in PARCOACH to prevent deadlocks at runtime consists in instrumenting all collectives starting from the first collectives that may deadlock in the program (cf **Section 5.3**). This instrumentation is conservative and prevents all possible deadlocks. However, the number of collectives instrumented can be very important in certain programs. An important perspective of improvement of PARCOACH is to minimize the number of collectives instrumented.

We illustrate the current instrumentation performed by PARCOACH and the perspective of improvement on the two examples of CFGs shown in **Figure 7.2**. In the figure, letters A, B, C, D, E represent collectives and collectives instrumented by PARCOACH are highlighted in bold.

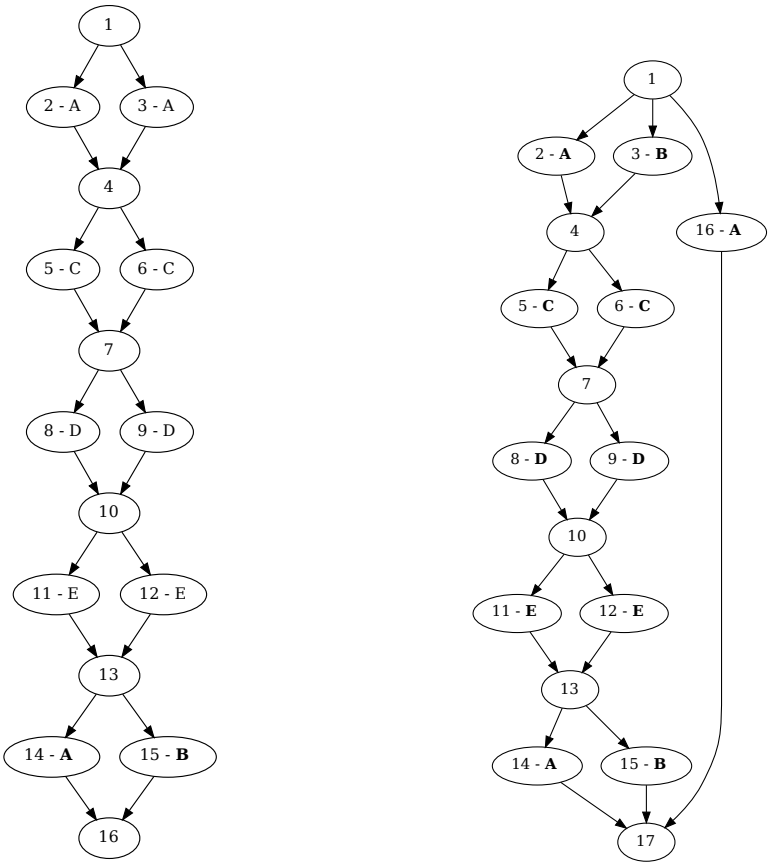
For the first CFG shown in **Figure 7.2a**, all processes will begin by executing the sequence of collectives $\{A, C, D, E\}$. Then depending on the conditional in node 13 some processes will execute the collective A whereas others will execute collective B. As the first collectives encountered during the CFG are called by all processes, PARCOACH will only instrument the collectives A and B in nodes 14 and 16. In this case the number of collectives instrumented is minimal as only collectives that may not be called by all processes are instrumented.

However, when the first collective encountered in the CFG may potentially not be called by all processes, PARCOACH instrument all collectives in the program. This situation is illustrated in **Figure 7.2b** where the first collective encountered by each process may be A or B depending on its execution path ($1-2$, $1-16$ or $1-3$).

In this example all collectives are instrumented. However, the number of collective instrumented is far from being minimal: Assuming that all processes begin by following the path $1-2-4$, then none of the collectives encountered during the path from node 4 to node 13 needs to be instrumented. Indeed, no matter the path taken by each process to go from node 4 to node 13, they will all execute the same collective sequence $\{C, D, E\}$.

In order to minimize the number of collectives instrumented, a solution would be to maintain the common language of remaining collectives to encounter by all processes and to only communicate when the next collective to execute is not a prefix of the common language.

The common language at the beginning of the program can be determined statically as a regular language over an alphabet representing all possible collectives. Thus, it is possible to only communicate when the common language L cannot be written as $L = u.L'$ with u a collective. In this case, before executing the next collective each process must share the language of collectives it still as to execute with all other processes. Two situations can arise:



(a) Only two collective instrumented. **(b)** All collectives instrumented.

Figure 7.2: Two example of CFGs with instrumented collective highlighted in bold.

- In the first case the intersection of the languages of all processes is null. It means that it is not possible that all processes execute the same sequence of collectives and the program must be stopped.
- In the second case the intersection is not null and the common language is updated.

In the CFG presented in **Figure 7.2b**, the common language is: $((A + B)CDE(A + B)) + A$. As this language does not have any prefix, all processes will communicate before executing the first collective. Assuming a process p_1 is in node 2, a process p_2 is in node 3 and a process p_3 is in node 16. Then the intersection of the languages of each process is null: $L = L_{p_1} \cap L_{p_2} \cap L_{p_3} = ACDE(A + B) \cap BCDE(A + B) \cap A = \emptyset$. Hence the program is stopped to prevent a deadlock. On the other hand if all processes execute the path from node 1 to node 2 then the intersection is determined as $ACDE(A+B)$. After executing the collective A each process can update the language of remaining collectives without communicating with others processes: $L = CDE(A + B)$.

Then, as CDE is a prefix of the common language L, no matter the path taken by each process to go from node 2 to node 13 they will all update locally the language of collectives they still have to execute but without communicating with other processes.

Finally, after executing the collectives C, D, E the language of remaining collectives is $L = A+B$

for each process. As this language does not contain a prefix, all processes will have to share their language with other processes before executing the next collective. If a process p_1 is in node 14 and a process p_2 is in node 15. Then the intersection of the languages is null and the program must be stopped: $L = L_{p_1} \cap L_{p_2} = A \cap B = \emptyset$. On the other hand if all processes are in node 14 then the intersection is not null and the program terminates normally after all processes execute collective A.

Going further, it would be possible to only communicate when the conditional leading to disjoint languages is multi-valued. For example, if the conditional in node 13 is single-valued then the processes do not need to share their language before executing the last collective (A or B). Indeed, if a process is in node 14, then all other processes are in the same node and will execute the same collective (A) since the conditional in node 13 is single-valued.

Nonetheless, to implement this solution it would be necessary to be able to calculate the intersection of the languages of a large number of processes at a low cost. Indeed, this solution is of no interest if the overhead of computing the intersection of the languages of all processes is higher than the overhead of instrumenting all collectives.

Bibliography

[1] M. Guest. The scientific case for hpc in europe 2012-2020. Technical report, PRACE, Brussels, Belgium, 2012. pages 9, 10

[2] J. D. OWENS. A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, 2005. pages 9

[3] https://www.top500.org. pages 10

[4] https://www.top500.org/green500/. pages 10

[5] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011. pages 11, 12

[6] http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015. OpenMP 4.5 Complete Specifications. pages 13

[7] http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. pages 13

[8] PGAS : Partitioned Global Address Space. http://www.pgas.org. pages 14

[9] George Almási, Paul Hargrove, Ilie Gabriel, and Tănase Yili Zheng. Upc collectives library 2.0. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011. pages 14

[10] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. pages 14

[11] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998. pages 14

[12] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. pages 14

[13] OpenAcc Directives for Accelerators. http://www.openacc.org. pages 14

[14] OpenCL - the open standard for parallel programming of heterogeneous systems. http://khronos.org/opencl. pages 15

[15] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. pages 16

[16] J.L. Lions. Ariane 5, flight 501, report of the inquiry board. Technical report, European Space Agency, 1996. pages 16

[17] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. pages 16

[18] Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, and Sigrid Eldh. Towards classification of concurrency bugs based on observable properties. In *Proceedings of the First International Workshop on Complex faUlts and Failures in LargE Software Systems*, COUFLESS '15, pages 41–47, Piscataway, NJ, USA, 2015. IEEE Press. pages 17

[19] Ganesh Gopalakrishnan, Paul D. Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A. Lethin, Koushik Sen, Stephen F. Siegel, and Armando Solar-Lezama. Report of the hpc correctness summit, jan 25-26, 2017, washington, dc. *CoRR*, abs/1705.07478, 2017. pages 18

[20] DDT debugger. https://www.allinea.com/products/ddt. pages 19

[21] LGDB. https://www.olcf.ornl.gov/software_package/cray-lgdb/. pages 19

[22] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997. pages 19

[23] Xuehai Qian, Koushik Sen, Paul Hargrove, and Costin Iancu. Sreplay: Deterministic subgroup replay for one-sided communication. pages 1–13, 06 2016. pages 19

[24] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, Nov 2007. pages 19

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, San Jose, CA, USA, 2004. pages 19, 92, 93, 111, 125

[26] Pierre Huchant, Marie-Christine Counilh, and Denis Barthou. Automatic opencl task adaptation for heterogeneous architectures. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, pages 684–696, New York, NY, USA, 2016. Springer-Verlag New York, Inc. pages 21, 131

[27] Pierre Huchant, Denis Barthou, and Marie-Christine Counilh. Adaptive Partitioning for Iterated Sequences of Irregular OpenCL Kernels. In *SBAC-PAD*, Lyon, France, September 2018. pages 21, 131

[28] P. Huchant, E. Saillard, D. Barthou, H. Brunie, and P. Carribault. Parcoach extension for a full-interprocedural collectives verification. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 69–76, Nov 2018. pages 21, 134

[29] Denis Barthou Pierre Huchant, Emmanuelle Saillard and Patrick Carribault. Multi-valued expression analysis for collective checking. In *Proceedings of the 25th International Conference on Euro-Par 2019: Parallel Processing*, 2019. pages 21, 134

[30] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2010. pages 25, 26, 133

[31] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. An extension of xcalablemp pgas lanaguage for multi-node gpu clusters. In Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Di Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, Stephen L. Scott, Jesper Larsson Traff, Geoffroy Vallée, and Josef Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, pages 429–439, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. pages 25

[32] Sylvain Henry, Alexandre Denis, Denis Barthou, Marie-Christine Counilh, and Raymond Namyst. Toward OpenCL Automatic Multi-Device Support. In *Proceedings of the 20th International Euro-Par Conference on Parallel Processing*, volume 8632 of *Euro-Par 2014*, pages 776–787. Springer International Publishing, Aug 2014. pages 25

[33] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag. pages 25

[34] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, United States, May 2013. pages 25

[35] Ashwin M. Aji, Antonio J. Peña, Pavan Balaji, and Wu chun Feng. MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL. *Parallel Computing*, 58:37 – 55, 2016. pages 25

[36] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Symp. on Microarchitecture*, MICRO 42, pages 45–55, New York, 2009. ACM. pages 26, 89

[37] P. Li, E. Brunet, F. Trahay, C. Parrot, G. Thomas, and R. Namyst. Automatic OpenCL Code Generation for Multi-device Heterogeneous Architectures. In *International Conference on Parallel Processing*, pages 959–968, 2015. pages 26, 71, 90

[38] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM Trans. Archit. Code Optim.*, 11(4):42:1–42:26, December 2014. pages 26

[39] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. pages 26

[40] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Principles and practice of Parallel Prog.*, PPoPP '11, pages 277–288, New York, 2011. ACM. pages 26, 71

[41] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *ACM Trans. Comput. Syst.*, 33(3):9:1–9:27, 2015. pages 26, 72, 89

[42] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. Orchestrating Multiple Data-Parallel Kernels on Multiple Devices. In *Parallel Arch. and Compilation Techniques*. IEEE, 2015. pages 26, 89, 133

[43] Borja Pérez, José Luis Bosque, and Ramón Beivide. Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems. In *Proceedings of the 9th Annual Workshop on GPGPU*, GPGPU '16, pages 42–51, New York, NY, USA, 2016. ACM. pages 26, 70, 90

[44] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. FinePar: Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. In *Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017. pages 26, 91

[45] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM. pages 34, 52, 53, 93, 101

[46] C. Oat, J. Barczak, and J. Shopf. Efficient spatial binning on the gpu, 2008. pages 38, 59

[47] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization*, pages 137–148, 2011. pages 38, 78

[48] Naohito Nakasato, Go Ogiya, Yohei Miki, Masao Mori, and Ken'ichi Nomoto. Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems. *CoRR*, abs/1206.1199, 2012. pages 40, 78

[49] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, Dec 1996. pages 50

[50] Daan Frenkel. *Understanding molecular simulation : from algorithms to applications*. Academic Press, San Diego, 2002. pages 58

[51] Pérez, Borja and Stafford, Esteban and Bosque, José Luis and Beivide, Ramón. Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. *The Journal of Supercomputing*, 73(1):330–342, Jan 2017. pages 70

[52] Raúl Nozal, Borja Pérez, and José Luis Bosque. Towards co-execution of massive data-parallel OpenCL kernels on CPU and Intel Xeon Phi. In *Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering*, 2017. pages 70

[53] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Intl Conf. on Supercomputing*, pages 149–160, New York, 2013. ACM. pages 71, 89

[54] R. Sakai, F. Ino, and K. Hagihara. Towards Automating Multi-dimensional Data Decomposition for Executing a Single-GPU Code on a Multi-GPU System. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 408–414, Nov 2016. pages 71

[55] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *ACM Intl Conf. on Supercomputing*, ICS '12, pages 341–352, New York, 2012. ACM. pages 71

[56] Prasanna Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Code Generation and Optimization*, pages 273–283. ACM, 2014. pages 71, 72, 90

[57] J. Lee, M. Samadi, and S. Mahlke. VAST: The illusion of a large memory space for GPUs. In *2014 23rd Intl Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 443–454, Aug 2014. pages 72

[58] Charles G Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 19(92):577–593, 1965. pages 77

[59] AMD. Ati stream software development ket (sdk) v2.1. http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx, 2010. pages 78

[60] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, May 2012. pages 78

[61] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic Optimization of Thread-coarsening for Graphics Processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 455–466, New York, NY, USA, 2014. ACM. pages 88

[62] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Intl Conf. on Compiler Construction*. Springer, 2011. pages 88

[63] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. Automatic OpenCL work-group size selection for multicore CPUs. In *Parallel Arch. and Compilation Techniques*, pages 387–397, 2013. pages 89

[64] Chih-Sheng Lin, Chih-Wei Hsieh, Hsi-Ya Chang, and Pao-Ann Hsiung. Efficient Workload Balancing on Heterogeneous GPUs using MixedInteger Non-Linear Programming. *Journal of Applied Research and Technology*, 12(6):1176 – 1186, 2014. pages 89

[65] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models. *IEEE Transactions on Computers*, 64(9), Sept. 2015. pages 89

[66] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *Computing Frontiers Conf.*, 2013. pages 90

[67] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. Heterogeneous parallel_for Template for CPU–GPU Chips. *International Journal of Parallel Programming*, Jan 2018. pages 90

[68] Borja Perez, Esteban Stafford, Jose Bosque, Ramon Beivide, Sergi Mateo, Xavier Teruel, Xavier Martorell, and Eduard Ayguade. Extending OmpSs for OpenCL Kernel Co-Execution in Heterogeneous Systems. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–8, 10 2017. pages 90

[69] J. A. Martínez, E. M. Garzón, A. Plaza, and I. García. Automatic tuning of iterative computation on heterogeneous multiprocessors with adithe. *J. Supercomput.*, 58(2):151–159, November 2011. pages 90

[70] Ester M. Garzón, J. J. Moreno, and J. A. Martínez. An approach to optimise the energy efficiency of iterative computation on integrated gpu–cpu systems. *The Journal of Supercomputing*, 73, 2016. pages 90

[71] Jie Shen, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G Simons. Glinda: a framework for accelerating imbalanced applications on heterogeneous platforms. In *Computing Frontiers Conf.*, page 14. ACM, 2013. pages 91

[72] Jie Shen, Ana Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. Improving performance by matching imbalanced workloads with heterogeneous platforms. 06 2014. pages 91

[73] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips. Workload partitioning for accelerating applications on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2766–2780, Sep. 2016. pages 91

[74] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. On-the-fly workload partitioning for integrated cpu/gpu architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 21:1–21:13, New York, NY, USA, 2018. ACM. pages 91

[75] GLPK (GNU linear programming kit), 2006. pages 93

[76] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, PPoPP, pages 194–204. ACM, 2007. pages 100, 125

[77] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Parcoach: Combining static and dynamic validation of mpi collective communications. *The International Journal of High Performance Computing Applications*, 28(4):425–434, 2014. pages 100, 102, 110, 134

[78] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Static Validation of Barriers and Worksharing Constructs in OpenMP Applications. In *International Workshop on OpenMP*, pages 73 – 86, Salvador, Brazil, September 2014. pages 100, 105, 125, 134

[79] Emmanuelle Saillard, Hugo Brunie, Patrick Carribault, and Denis Barthou. PARCOACH Extension for Hybrid Applications with Interprocedural Analysis. In *9th International Workshop on Parallel Tools for High Performance Computing*, pages 135 – 146, Dresden, Germany, September 2015. pages 100, 104, 134

[80] Julien Jaeger, Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH. In *EuroMPI*, 2015. pages 100, 134

[81] Yuan Lin. Static nonconcurrency analysis of openmp programs. In Matthias S. Mueller, Barbara M. Chapman, Bronis R. de Supinski, Allen D. Malony, and Michael Voss, editors, *OpenMP Shared Memory Parallel Programming*, pages 36–50, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. pages 101

[82] MILC. http://www.physics.utah.edu/~detar/milc/milc_qcd.html, 2016. pages 111, 126

[83] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005. pages 111, 126

[84] Alexander J. Ropelewski, Hugh B. Nicholas, Jr, and Ricardo R. Gonzalez Mendez. MPI-PHYLIP: Parallelizing Computationally Intensive Phylogenetic Analysis Routines for the Analysis of Large Protein Families. *PLOS ONE*, 5(11):1–8, 11 2010. pages 111, 126

[85] CORAL AMG. https://asc.llnl.gov/CORAL-benchmarks/Summaries/AMG2013_Summary_v2.3.pdf, 2013. pages 111, 126

[86] High-Performance Linpack benchmark. http://www.netlib.org/benchmark/hpl/, 2016. pages 111, 126

[87] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009. pages 111, 126

[88] NERSC IOR. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/ior/, 2016. pages 111, 126

[89] Hydro Benchmark. https://github.com/HydroBench/Hydro, 2017. pages 111, 126

[90] 2016. NASPB site: https://www.nas.nasa.gov/publications/npb.html. pages 111, 126

[91] *Cori*, 2017. http://www.nersc.gov/users/computational-systems/cori/. pages 114

[92] Alexander Aiken and David Gay. Barrier inference. In *Proc. ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, POPL, pages 342–354. ACM, 1998. pages 117

[93] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–34, Sept 2008. pages 120

[94] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proc. Int. Conf. on Comp. Construction*, CC, pages 265–266. ACM, 2016. pages 120, 121, 124, 125, 155

[95] Ben Hardekopf and Calvin Lin. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *Proc. Symp. on Code Generation and Optimization*, CGO, pages 289–298, 2011. pages 120

[96] Y. Sui, D. Ye, and J. Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Trans. Softw. Eng.*, 40(2):107–122, 2014. pages 120

[97] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.*, 49(6):259–269, 2014. pages 124

[98] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, PLDI, pages 87–97. ACM, 2009. pages 124

[99] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, 2001. pages 124

[100] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977. pages 124

[101] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'98, pages 365–377, 1998. pages 124

[102] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. ACM Conf. on Comp. and Communications Security*, CCS, pages 116–127, 2007. pages 124

[103] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, 2006. pages 124

[104] Ignacio Laguna and Martin Schulz. Pinpointing Scale-dependent Integer Overflow Bugs in Large-scale Parallel Applications. In *Proc. Conf. for High Perf. Comp., Networking, Storage and Analysis*, SC, pages 19:1–19:12, 2016. pages 124

[105] Ding Ye, Yulei Sui, and Jingling Xue. Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis. In *Proc. IEEE/ACM Symp. on Code Generation and Optimization*, CGO, pages 154:154–154:164, 2014. pages 124

[106] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–53, 1991. pages 124

[107] Asia Slowinska and Herbert Bos. Pointless Tainting?: Evaluating the Practicality of Pointer Tainting. In *Proc. ACM European Conf. on Comp. Systems*, EuroSys'09, pages 61–74, 2009. pages 124

[108] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Strict Control Dependence and Its Effect on Dynamic Information Flow Analyses. In *Proc. Symp. on Software Testing and Analysis*, ISSTA'10, pages 13–24, 2010. pages 124

[109] Cristina Cifuentes and Bernhard Scholz. Parfait: Designing a Scalable Bug Checker. In *Proc. Workshop on Static Analysis*, SAW'08, pages 4–11, 2008. pages 124, 125

[110] Stephen F. Siegel and Timothy K. Zirkel. Automatic Formal Verification of MPI-based Parallel Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 309–310, New York, NY, USA, 2011. ACM. pages 125

[111] Patrick Ohly and Werner Krotz-Vogel. Automated MPI Correctness Checking: What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, South Lake Tahoe, California, USA, 2007. pages 125

[112] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de de Supinski, Martin Schulz, and Greg Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High*

Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. pages 125

[113] Tobias Hilbrich, Bronis R. de Supinski, Fabian Hänsel, Matthias S. Müller, Martin Schulz, and Wolfgang E. Nagel. Runtime MPI Collective Checking with Tree-based Overlay Networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 129–134, New York, NY, USA, 2013. ACM. pages 125

[114] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A Graph Based Approach for MPI Deadlock Detection. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 296–305, New York, NY, USA, 2009. ACM. pages 125

[115] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007. pages 125

[116] Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In *PVM/MPI*, pages 18–27. Springer, 2004. pages 125

[117] Hongyi Ma, Steve R. Diersen, Liqiang Wang, Chunhua Liao, Daniel Quinlan, and Zijiang Yang. Symbolic Analysis of Concurrency Errors in OpenMP Programs. In *PARCO*, volume 00 of *ICPP*, pages 510–516, 2013. pages 125

[118] Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao. Languages and compilers for parallel computing. chapter Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers, pages 95–109. Springer-Verlag, Berlin, Heidelberg, 2008. pages 125

[119] The GNU Compiler Collection. https://gcc.gnu.org. pages 125

[120] The Intel Compiler. https://software.intel.com/en-us/intel-compilers. pages 125

[121] U. Banerjee, B. Bliss, Z. Ma, and P. petersen. Unraveling data race detection in the intel thread checker. In *Int'l. Symp. on Computer Architecture*, ISCA, 2008. pages 125

[122] Christian Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 669–676. IOS Press, 2007. pages 125

[123] Intel Inspector XE. https://software.intel.com/en-us/intel-inspector-xe, 2017. pages 125

[124] Thread Analyzer user guide. https://docs.oracle.com/cd/E19205-01/820-0619/index.html, 2010. pages 125

[125] https://developer.nvidia.com/cuda-memcheck, 2017. Cuda Memcheck, NVidia. pages 126

[126] James Price and Simon McIntosh-Smith. Oclgrind: An Extensible OpenCL Device Simulator. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL'15, pages 12:1–12:7. ACM, 2015. pages 126

[127] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 113–132. ACM, 2012. pages 126

[128] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic Testing of OpenCL Code. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing: 7th International Haifa Verification Conference*, pages 203–218. Springer Berlin Heidelberg, 2011. pages 126

[129] Ebnenasir Ali. UPC-SPIN: a framework for the model checking of UPC programs. In *Proceedings of the fifth conference on partitioned global address space programming models*, PGAS'11, 2011. pages 126

[130] James Coyle, Indranil Roy, Marina Kraeva, and Glenn R. Luecke. UPC-CHECK: a scalable tool for detecting run-time errors in Unified Parallel C. *Computer Science - Research and Development*, 28(2):203–209, 2013. pages 126

[131] Samuel Williams. Implementation and optimization of miniGMG-a compact geometric multigrid benchmark. 2014. pages 126

[132] Haoqiang Jin, Robert Hood, and Piyush Mehrotra. A practical study of UPC using the NAS Parallel Benchmarks. In *PGAS*, page 8, 2009. pages 126

[133] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009. pages 126

List of Figures

1.1 Exponential growth of supercomputing power as recorded by the TOP500 list. . . . 11
1.2 MPI Collective Operations. . . . 14

2.1 Pseudocode of a parallel computation kernel. . . . 26
2.2 Pseudocode of an application with an iterated sequence of *m* kernels. . . . 27
2.3 Single device iterative application with *m* kernels written in different languages. . . . 28
2.4 Parallelization of the SOTL application. . . . 29
2.5 Transformation of a single-device iterative application into a multi-device application. 30
2.6 Partitioning Input Arrays. The real region required onto each device is highlighted in red. With a memory region analysis, only the required data is transferred from the host to each device. . . . 31
2.7 Partitioning Output Arrays. The partial region written by each device is highlighted in red. With a memory analysis only the partial regions written by each device are transferred to the host. . . . 31
2.8 Splitting data when there is a data dependency between two kernels. . . . 32
2.9 Stencil Partitioning. . . . 33
2.10 1D Stencil kernel. . . . 34
2.11 2D Stencil kernel. . . . 35
2.12 Memory regions of buffer `B` written by each sub-kernel when the stencil2D kernel is partitioned on 2 devices by splitting the second dimension of the NDRange. The gray area shows the overapproximation obtained by an interval analysis but in fact only elements in darker gray are actually written by each sub-kernel. . . . 36
2.13 Memory regions of buffer `B` written by each sub-kernel when the stencil2D kernel is partitioned on 2 devices by splitting the first dimension of the NDRange. The gray area shows the overapproximation obtained by an interval analysis but in fact only elements in darker gray are actually written by each sub-kernel. . . . 37
2.14 Spatial Binning kernel. . . . 37
2.15 (a) and (b): Impact on performance of architectural heterogeneity on AESEncrypt and EP benchmarks. Performance is given as an average time per work-group, partitioning ratio as a percentage of the total number of work-groups. (c) Impact on performance of the offset (starting index) for SpMV kernel, with a fixed partitioning ratio of $1/4$. (d) Impact of iteration number on performance for OTOO application. Partitioning ratios are set to $1/4$ for all GPUs, and offset is fixed on all devices. . . . 39
2.16 Impact on amount of data to transfer of different partitioning strategies. . . . 40
2.17 Partitioning a sequence of kernels. . . . 42

2.18 Framework Overview. At load time when a kernel is loaded, it is analysed by the *Kernel Analyzer* and a partition-ready kernel is generated by the *Kernel Transformer*. At runtime before a kernel is executed, the *Dynamic Partitioner* determine a partitioning for the kernel based on previous iteration if any and the *Buffer Manager* handle the necessary data transfers before sub-kernels execution. 43
2.19 Splitting one dimension of a 3D NDRange to distribute the parallel iteration space onto 4 devices. 44
2.20 General algorithm of the dynamic adaptation of a single-device iterative application with *m* kernels to *n* heterogeneous devices. 47

3.1 Indirection array annotation. In this example, the input buffer `A` is indirectly accessed through buffer `IA` and `IA` is annotated as increasing. 51
3.2 SpatialBinning kernel Partitioning. 53
3.3 Write Region Analysis with Replication. 56
3.4 Overview of SOTL. 58
3.5 Subdivision of the 3D space of the domain into cells. 59
3.6 SOTL: Cells array. 60
3.7 SOTL: Sorting atoms. 61
3.8 *force* kernel from SOTL. 62
3.9 Partitioning arrays from the *force* kernel. 65
3.10 *count* kernel from SOTL. 65
3.11 Partitioning arrays from the *count* kernel. 66
3.12 *sort* kernel from SOTL. 67
3.13 Partitioning arrays from the *sort* kernel. 68
3.14 Performance obtained when partitioning SOTL on multiple GPUs. 70

4.1 Formulations of the partitioning problem. 76
4.2 Performance of AESEncrypt, EP, MonteCarlo, OTOO, SPMV and some Polybench on CONAN and HAPPYCL. Original codes run only on one device. UNIFORM and ADAPTIVE are using sub-kernels automatically obtained by our method. 79
4.3 Speedup per iteration of EP and AESEncrypt. 80
4.4 Performance of OTOO executed on CONAN (3GPUs+CPU) for 60 iterations. 80
4.5 Amount of data to transfer to device 2 before executing kernel *k* depending on the partitioning of *h* and *k* when these kernel are partitioned onto 3 devices. 83
4.6 Linear system solved at each iteration with the ADAPTIVE W/ COMM strategy. 85
4.7 Overall Results Obtained. 87
4.8 Time taken by computation and transfer when SOTL St is partitioned on CONAN using ADAPTIVE W/O COMM strategy versus ADAPTIVE W/ COMM strategy. 87
4.9 Total time per iteration when SOTL Dyn is partitioned on CONAN. 88
4.10 Splitting a NDRange into two sub-NDRanges. 92

5.1 MPI examples of control-flow divergences that may lead to the execution of different sequences of collectives by different processes. 101
5.2 Example of Control Flow Graph. 102
5.3 A simple example and its instrumentation. 103
5.4 Examples of MPI and OpenMP codes. 105
5.5 MPI Code 4 functions CFG (left) and the corresponding PPCFG (right). 107

5.6 MPI Code 1 and OpenMP Code 3 PPCFG. 109
5.7 MPI example and its instrumentation. 110
5.8 Number of warnings added and removed with PARCOACH using the full-interprocedural method compared to PARCOACH using the intraprocedural analysis. 113
5.9 Number of conditionals added and removed with PARCOACH using the full-interprocedural method compared to PARCOACH using the intraprocedural analysis. 113
5.10 Compile-time overhead using the full-interprocedural analysis (ratio between the PARCOACH analysis time and the total compilation time). 113
5.11 Execution-Time of Hydro with and without runtime verification (domain size = 500x500, nstepmax=200). 114

6.1 Examples of collective issues. 119
6.2 Enhanced SSA form of the MPI code **Figure 6.1b** and its corresponding PDCG. . . 123
6.3 Percentage of warnings and conditionals filtered by our analysis. 100% means that the analysis has shown the program is free of collective error. The total number of filtered warnings and conditionals is given at the top of each bar. 127
6.4 Examples of MPI codes. 128

7.1 Annotation Example. 135
7.2 Two example of CFGs with instrumented collective highlighted in bold. 136

List of Tables

4.1 Applications and Benchmarks Description. 78
4.2 Applications and Benchmarks Description. 86
4.3 Framework Components. 92

5.1 Applications and Benchmarks Statistics. 112
5.2 Number of warnings reported and conditionals responsible for a collective error for both intraprocedural and full-interprocedural analyses. 112

6.1 Building Rules for Instructions in the PDCG. Value Flow Dependence rules are based on SVF [94] with our differences highlighted in red. Optimization rules eliminate spurious dependences and Collective Checking rule connects collectives to the conditionals governing their execution. 121
6.2 Applications and Benchmarks Characteristics. OMP=OpenMP. 127
6.3 Multi-valued detection comparison between PARCOACH, SVF and Parfait, both combined with collective deadlock detection. FP = false positives, FN = false negative. 129